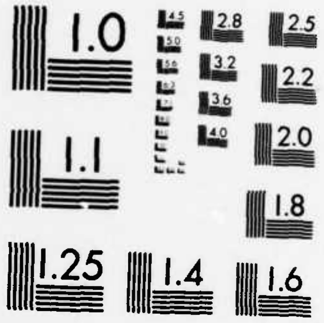
AD-A139 104 A METHODOLOGY FOR BENCHMARKING RELATIONAL DATABASE MACHINES(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P R STRAWSER JAN 84 NPS52-84-004 1/3

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(2)

NPS52-84-004

NAVAL POSTGRADUATE SCHOOL
Monterey, California

A METHODOLOGY FOR BENCHMARKING
RELATIONAL DATABASE MACHINES

Paula R. Strawser

January 1984

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research, Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research, and in part by the contract N00014-84-WR-2408 of the Office of Naval Research.

This report was prepared by:

Paula R. Strawser

Paula R. Strawser
Adjunct Professor of Computer Science

Reviewed by:

[illegible]

D. K. HSIAO, Chairman
Department of Computer Science

Released by:

[illegible]

KNEALE T. MARSHALL
Dean of Information and Policy Science

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-84-004	2. GOVT ACCESSION NO. AD-A139104	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A METHODOLOGY FOR BENCHMARKING RELATIONAL DATABASE MACHINES		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Paula R. Strawser		8. CONTRACT OR GRANT NUMBER(s) N-00014-84-WR-24058
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Department of Computer Science (Code 52) Monterey, CA 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N: RR014-08--01 N0001484WR24058
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE January 1984
		13. NUMBER OF PAGES 236
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) database machine, database computer, performance evaluation, benchmarking		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The thesis presents a methodology for benchmarking relational database machines. It provides a standard for benchmarking relational database machines and relational software database management systems. As a methodology, a collection of methods, tools, and procedures is included. The methodology is based on a three-level hierarchy of models. At the lowest level is the machine model. The machine is modeled as a high-level language architecture, i.e. as a machine which executes a relational query language. This approach

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Cont'd

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

gives a degree of machine-independence. The next level in the hierarchy is the database model. The database is modeled as a synthetic database consisting of twelve relations of varying tuple width and cardinality. The relations are based on a standard tuple template. This approach gives us database independence, i.e., the database model is independent of any real-world database. The highest level in the hierarchy is the applications model. We present two applications models.

The first applications model consists of a collection of basic benchmarks. The idea of the basic benchmarks is to measure the best-case performance of the machine. The methodology includes a method for constructing the basic benchmark jobs, a procedure for executing the basic benchmark jobs, detailed specification of the benchmarks to be executed, and methods for interpreting the measurements from the benchmarks.

The basic benchmark jobs are constructed using one of two simple job models. The benchmarks are executed in single-user mode, in a carefully controlled environment. Detailed specifications are given for twenty-five basic benchmarks. There are seven benchmarks which measure machine performance for ordering and aggregation, three which concern projection, nine which measure performance for a variety of join operations, and two which measure the performance of the machine for update, delete, and insert operations.

A general procedure for analyzing the measurements is supplemented by the detailed instructions given with each benchmark specification. The analysis of the measurements results in a classification of the benchmark queries. The classification scheme is based on the resources used in execution. There are three classes of queries: access-intensive, processor-intensive, and overhead-intensive. Three simple performance indices, one for each classification, are developed to express the performance of the machine for each of these query classes.

The classification of queries and the performance indices from the basic benchmarks are used to develop the second applications model. The second applications model is a multiple-user benchmark model. The idea of the multiple-user benchmarks is to compare the performance of the machine with multiple users to the best-case performance measured in the basic benchmarks. The data for each query classification collected from the basic benchmarks and the performance indices from the basic benchmarks are used to construct standard jobs for multiple-user benchmarks. A standard access-intensive job, a standard processor-intensive job, and a standard overhead-intensive job are constructed. Performance indices which reflect the actual performance of the machine with multiple users are developed. In addition, a set of comparative indices are developed to reflect actual vs. ideal performance.

Experiments in an application of the methodology are reported. Nine of the basic benchmarks are included in the experiments. The results show that the methodology provides useful information about the performance of the benchmarked machine. The calculation of the performance indices and the construction of standard jobs for multiple-user benchmarks are illustrated using the measurements collected in these experiments.

The contributions of the work to the body of knowledge of database machines, database systems, and performance evaluation are discussed. Directions for further research are identified.

S/N 0102- LF- 014- 6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

For Raoul Biondo

ACKNOWLEDGEMENTS

The work reported in this thesis, begun in the spring of 1982, was supported by Contract N00014-84-WR-24058, Office of Naval Research.

I would like to extend my thanks to the following people:

David K. Hsiao, for the opportunity and the guidance;

Doris Mleczko and coworkers at DPSCWEST, for excellent support during experimentation;

Bob Bogdanowicz, Michael Crocker, Curt Ryder, and Vince Stone, for late nights spent collecting experimental data and for moral support;

Steve Fuld and coworkers at Amperif Corporation, for technical support;

Steve Demurjian, for his careful reading, constructive criticism, and sustained interest;

Doug Kerr and Karsten Schwan, for their evaluation of the work and suggestions for improvement;

the faculty and staff of The Ohio State University;

the staff at Naval Postgraduate School, for excellent technical and clerical support;

my family, for believing in me;

my friend Cler, without whose encouragement I might never have dared graduate school.

VITA

October 26, 1948. . . .	Born, Morgantown, WV
1970.	B.S.B.A., West Virginia University, Morgantown, WV
1980-1983	Graduate Assistant, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH
1982.	M.Sc., The Ohio State University, Columbus, OH
1983.	Adjunct Research Instructor, Naval Postgraduate School, Monterey, CA

PUBLICATIONS

"The Implementation of a Multi-Backend Database System (MDBS): Part I - An Exercise in Database Software Engineering", Advanced Database Machine Architectures, Prentice-Hall, 1983, pp. 300-326, multiple authors.

"The Implementation of a Multi-Backend Database System (MDBS): Part II - The Design of a Prototype MDBS", Advanced Database Machine Architectures, Prentice-Hall, 1983, pp. 327-385, multiple authors.

"Benchmarking the RDM-1100 Relational Database Machine: A Preliminary Report Part III", Proceedings of the USE Inc. Spring Conference, Volume 1, pp. 177-195, multiple authors.

"Experiments in Benchmarking Relational Database Machines", Database Machines, Springer-Verlag, 1983, pp. 106-134, multiple authors.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Database Systems.
Professor David K. Hsiao

Studies in Computer Architecture.
Professor Frank Crow

Studies in Software Engineering.
Professor Stuart H. Zweben

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Database machines are coming of age. In 1981, the first commercial product entered the marketplace. Competing machines will become available in the near future. As the number of alternatives increases, the decision to buy or not to buy a database machine becomes increasingly complicated. The consumer must decide not only whether a database machine offers advantages, but which database machine best suits his/her applications.

Among the advantages claimed for database machines are host independence, support for multiple, dissimilar hosts, increased reliability, and enhanced security [Mala82] [Hsia82]. Since the database machine is independent of the host, growth in the database can be accommodated by augmenting the resources of the database machine, without change in the host configuration. Databases can be shared among multiple, dissimilar hosts through the database machine. Reliability is enhanced by implementing some data management functions in firmware and hardware rather than in software. Security is enhanced both by the physical separation of the database from the host, and by limiting access to the database machine to authorized users.

But perhaps the most important claim is that database machines offer a price/performance advantage. The claim is that the user will get more performance per database machine dollar than per dollar invested in a software database management system (DBMS) -- more "bang for the buck".

To prove the price/performance advantage claim, we must quantify the costs and measure the performance of both software DBMSs and database machines. This is a problem of considerable complexity. To reduce the problem to manageable dimensions, we divide the problem into two parts: price/performance evaluation of software DBMSs and price/performance evaluation of database machines. We can apply the divide-and-conquer strategy again, and further subdivide each problem into two parts: quantification of costs and performance evaluation. By concentrating our efforts on one of these subproblems, we can contribute to the overall solution.

1.1. Defining the Problem

We select as our basic problem the performance evaluation of database machines. The scope of this problem is too broad to be addressed in a single thesis. We are particularly interested in the evaluation of alternative database machine architectures and of variations in database machine configurations. There is a need for a performance evaluation standard for database machines. With this need in mind, let us continue to apply the divide-and-conquer strategy to limit the scope of the problem.

1.1.1. Modeling vs. Measurement

There are two general classes of techniques for evaluating the performance of computer systems, modeling techniques and measurement techniques [Ferr78]. In the former, a model of the system is proposed. Performance information can be derived from the model either through analytic methods or through simulation. In the latter class of techniques, performance information is obtained from direct measurements of the system.

When a modeling technique is used, the values which characterize system performance are based on estimates and/or measurements of the performance of an actual system. These techniques are especially useful when designing systems. Various designs may be modeled and the results compared. A problem common to all modeling techniques is the accuracy of the model. In order to be confident that the model is accurate, it is desirable to compare the results to actual measurements of the system.

We have chosen to use a measurement technique for the following reasons. First, we expect to be comparing existing machines. Thus a measurement technique is valid. Second, measurement techniques are attractive because they are both accurate and credible [Saue81]. Third, we can concentrate on the workload characterization problem instead of the problem of the accuracy of the model of the machine. Finally, as an area for further research, the workload characterization which we will develop can be used to characterize workload for a model, and the results can be used to tune a model. Thus, the scope of our problem is performance measurements for database machines.

1.1.2. Why Benchmarks?

In measurement studies as well as in modeling studies, the workload must be characterized. Measurement studies may be performed under real workload conditions, or with an artificial workload. The artificial workload is called a benchmark.

The term benchmark originates from the markers used by surveyors in establishing common reference points for their measurements. For example, Mount Diablo (a mountain east of San Francisco) is used as a reference point in surveying much of northern California due to its long-range visibility. Thus the connotation of the word benchmark is a measurement against some standard.

In computer performance evaluation, a benchmark is "a set of executable instructions which may be used to compare the relative performance of two or more computer systems." Benchmarking is the process of conducting a set of controlled experiments to collect performance measurements which may be used to compare two or more systems. [Morr82] Benchmarks are often used for qualitative evaluation as well as for quantitative evaluation. That is, a benchmark might be used not only to determine how quickly a system accomplishes a given task, but also to evaluate the quality of the services which the system provides. For example, a qualitative benchmark for a database machine might be used to evaluate features such as backup and recovery, report writers, and interactive interfaces. Our purpose will be quantitative evaluation, although some qualitative information may be available as a byproduct.

Benchmarks may be constructed by selecting portions of a real workload, or may be based on synthetic jobs [Rose76]. Constructing benchmarks for database machines is a multi-dimensional problem. There is the problem of host workload characterization and the problem of database machine workload characterization. When characterizing the database machine workload, it is not sufficient to model only the applications. The database must also be modeled. The performance of a machine for a given query will vary depending on the content of the database. Clearly the performance measurements for retrieval of a large number of records from a database will differ from the results for retrieval of a single record. We must also consider that there will be cases where neither the database nor the applications will exist at the time the selection of machines is being made.

Let us further refine the scope of our problem to _benchmarking database machines_. In order to be able to control the structure of the database as well as the applications, we choose to use an executable artificial workload model based on synthetic jobs and a synthetic database. This technique allows us to provide information in those cases where the database does not exist in machine-readable form and/or the applications have not been developed. We also avoid the privacy and security problems inherent in working with real database. Most important, the technique provides generality. If the model is carefully constructed and guidelines for interpretation are provided, the benchmark results will be applicable to a variety of applications.

1.1.3. Which Database Machines?

A database machine (DBM) is a special-purpose hardware/software architecture designed for performing the data management tasks. The DBM is configured as a backend processor to one or more (possibly dissimilar) hosts, or as a node in a network of machines. The sole function of the DBM is to perform the data management functions in response to queries or transactions transmitted from the host or hosts.

Some database machines have been designed for special purposes, for example text retrieval. The majority of database machines, however, have been designed to support a variety of applications using formatted databases. We will limit the scope of this study to those DBM architectures which support formatted databases. In order to further simplify the problem, we restrict our study to those DBMs which support the relational model. A survey of existing and proposed DBM architectures indicates that most support the relational model or some subset of that model. Therefore, this restriction to relational machines is not a severe limitation. So the scope of our problem is refined to benchmarking relational database machines.

1.1.4. What's in a Methodology?

Peter Freeman writes:

> "There are three types of organized software development technology: methods (ways of doing something), tools (objects such as programs, languages, or documentation forms that help us use a method), and methodologies (collections of methods and tools along with the management and human-factors procedures necessary to their application.)" [Free79]

This delineation of methods, tools, and methodologies is applicable in performance evaluation as well as in software development. It will not be sufficient merely to develop benchmarks for relational database machines. We must also provide the tools and procedures for conducting the experiments and for implementing the benchmarks. The scope of our problem is now defined as <u>a methodology for benchmarking relational database machines</u>.

1.2. The Goal

Our goal is to develop a methodology for benchmarking relational database machines, for the purposes of comparative evaluation of database machine architectures and varying database machine configurations. The methodology will be applicable to a wide range of machines, databases and applications. The basis of the methodology is a synthetic model of the database and the applications. The methodology includes the following:

1) Methods for characterizing workload;

2) A tool for generating synthetic databases;

3) A scheme for classifying queries;

4) A method for calculating performance indices;

5) Methods for analyzing performance measurements;

6) Methods for constructing basic benchmark jobs;

7) Procedures for executing basic benchmark jobs;

8) Methods for constructing multiple-user benchmarks;

9) Procedures for executing multiple-user benchmark job mixes;

1.3. Organization of the Thesis

The thesis is loosely organized along the lines of the methods, tools, and procedures listed in the previous section. We treat the design issues and the design decisions for each method in a separate chapter or chapters. In Chapter 2 we discuss the workload characterization methods. In Chapter 3 we propose the query classification scheme. Chapter 4 is a discussion of the method for calculating performance indices. The methods for constructing basic benchmark jobs and the procedures for executing these jobs are detailed in Chapter 5. The method for interpreting basic benchmark results is described in Chapter 6. In Chapter 7 we describe the standard set of basic benchmarks. A discussion of the methods for constructing multiple-user jobs is included in Chapter 8. In Chapter 9, we show some results from a partial application of the methodology, and illustrate the method of interpreting results. A review of the contributions of this research, and directions for further research are set forth in Chapter 10. The tool for generating synthetic databases is described in Appendix A.

2. WORKLOAD CHARACTERIZATION

The workload of a computer system is defined as the set of all inputs the system receives from its environment [Ferr78]. In Chapter 1 we specified that an artificial workload model based on a synthetic database and a set of synthetic applications will be used. An artificial workload model "consists of basic components purposely devised to be used to load a real system or a model of it " [Ferr83]. But we must consider that we are dealing with two or more computer systems: at least one host machine and a database machine. We must construct not one but two workload models. In the following sections we first discuss the characteristics of a workload model. We then describe our host workload model and our database machine workload model. Our workload models will be evaluated in Chapter 9.

2.1. Characteristics of a Workload Model

Ferrari lists eight characteristics of a workload model: representativeness, flexibility, reproducibility, system-independence, simplicity of construction, compactness, usage costs, and compatibility [Ferr78]. He defines these characteristics with reference to evaluating a model of a real workload. We will not model a real workload. Instead, we will develop a workload model which has relevance for a wide variety of machines, databases, and applications. We will extend Ferrari's definitions to fit this more general model.

2.1.1. Representativeness Extended to Generality

Ferrari defines representativeness as the accuracy of the model. The workload must accurately reflect the characteristics of some real workload. Our goal is broader than modeling a specific workload. We aim to provide a methodology which is applicable to a wide range of databases and applications. Let us extend the definition to generality. Generality requires that the benchmark results be representative of a wide variety of database and applications, and be valid across differing database machine architectures. To achieve generality, our model must be machine-independent, database-independent, and application-independent, as well as representative.

2.1.2. Flexibility

Ferrari defines flexibility as "the possibility of easily and inexpensively modifying a model to reflect variations in the real work load." In the context of our methodology, flexibility is most important in the applications model for multiple-user benchmarks. So, let us specialize the definition of flexibility to the possibility of easily and inexpensively selecting instruction mixes which approximate a real workload.

2.1.3. Reproducibility and System-Independence, with Comparability

Reproducibility refers to the ability to produce consistent results when repeating experiments. System-independence is "the extent to which a model can be transported from system to system while still remaining sufficiently representative" [Ferr78]. While we do not need to modify these definitions, we do need to define another characteristic for our model. This is comparability, the ability to produce results which can be meaningfully compared from system to system.

2.1.4. Simplicity of Construction, Compactness, Usage Costs, and Compatibility

Simplicity of construction includes the cost and complexity of gathering the information required to design the workload model and to make it operational. Compactness is related to the degree of detail. Usage costs are the costs of constructing and implementing the workload model. Compatibility refers to the compatibility of the model and the system. For example, a model which is to drive a real system must be in a form which is executable on that system [Ferr78]. We can accept these definitions without change. It should be noted that there are tradeoffs between these characteristics and representativeness.

2.2 The Host Workload Model

In a production environment, the processes of the database machine interface and the processes of database machine users will share the resources of the host machine with other processes. The portion of the host machine resources consumed by DBM-related processes will vary from host to host. The sharing of resources with other processes will affect the apparent performance of the database machine. Response times will be inflated by time spent waiting for host machine resources.

One way to model this is to assign some overhead to every DBM-related process. However, how can we estimate the overhead? In order to estimate the overhead, we need a model of the non-DBM-related host workload. The model will vary from host machine to host machine and from installation to installation. Constructing the models is an expensive and complex task.

Let us take a simpler approach. We will model the host workload as consisting only of those processes necessary to support the interface with the database machine and user processes which are interacting with the database machine. We will be measuring the <u>best-case interaction</u> between the host and the database machine. The measurements will not be inflated by delay at the host machine.

This approach is valid within our framework. We are looking at comparative evaluation of database machine architectures. All machines will be benchmarked using the same host workload model. Results can be compared without adjustment for effects of the host workload. Of course, the performance of the host interface will vary from machine to machine. However, since the performance of the host interface, from our perspective is an integral component of elapsed time for query execution, we will make no adjustment for different interfaces.

As a side-effect of measuring the best-case interaction between the host and the database machine, we have the capability to drive the database machine workload at the maximum arrival rate. An additional advantage of this strategy is that it facilitates measurements of the host resources required to support the database machine. Measurements of host activity during benchmark execution, when the only active processes are operating system processes and the database machine interface processes, can be used to identify the host workload required to support the database machine.

2.3. The Database Machine Workload Model

Our database machine workload model is actually composed of three models: a model of the machine, a model of the database and a model of the applications. These models are hierarchically dependent, as illustrated in Figure 1. At the lowest level is the model of the machine. At the next level is the model of the database, which depends on the machine model. At the highest level is the applications model, which depends on the database model. To achieve a benchmarking methodology which is relevant for a wide range of databases and applications, we must construct our model to be machine-independent, database-independent, and application-independent.

In the next three sections, we describe the machine model and the database model. The applications model is described briefly. More details of the applications model will be presented Chapters 5, 7, and 8.

2.3.1. The Machine Model and Machine-Independence

To achieve machine-independence, the benchmarks must be constructed without bias toward any particular architecture. We have

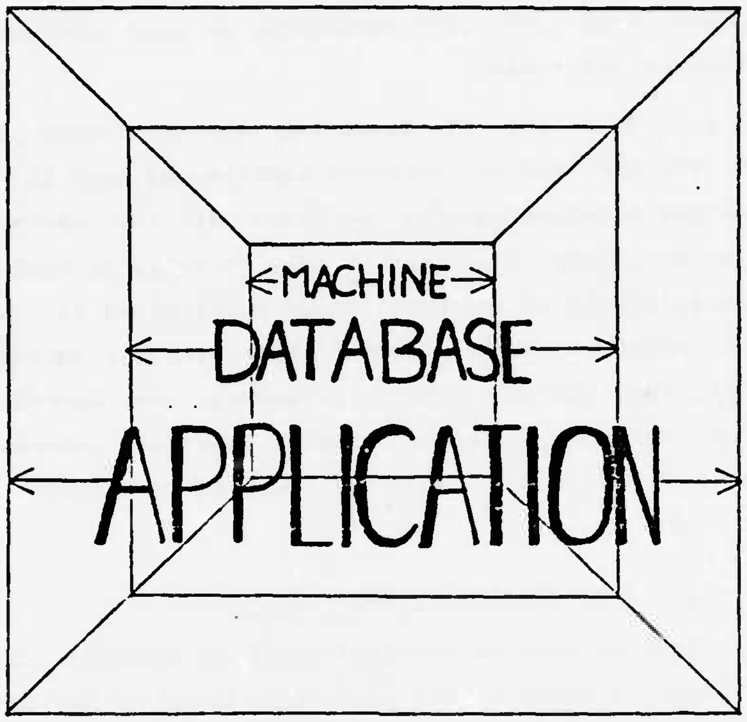


Figure 1: Hierarchically Dependent Models

targeted our methodology for relational database machines. An examination of existing and proposed architectures shows that the development of a general model of database machine architectures is infeasible. (See Chapter 6.) The physical architectures vary widely. The issue is further complicated by the varying distribution of functionality between hardware and software.

[Hayn82] presents the idea of database machines as a subset of high-level-language computers. The concept of a high-level-language computer is that the machine is designed to support the constructs of some high-level language. The relational database machine, then, is a machine designed to support the relational query language. Although relational query languages vary, they all support the operations of the relational model. These include the basic operations of any data management system, retrieval, update, deletion, and insertion. Also included are the special relational operations, selection, projection, join, and union, as well as the commonly available operations of aggregation and ordering. We will model the database machine as a <u>high-level language computer</u> which supports the relational query language.

However, the benchmarking methodology requires that we have some knowledge of the particular architecture being benchmarked in order to make the benchmarks more realistic. These kinds of knowledge can be introduced as parameters of the database machine model. We now extend our basic model with a set of variable parameters which represent:

a) The size of the basic unit of data management or block, i.e, the smallest unit of access to the devices in the database store;

b) The total size of available primary memory, if any, or the total volume of data which can be simultaneously accessed from the database store;

c) The set of mechanisms by which the user can control distribution of data across the devices in the database store; and

d) The set of index structures and access methods supported.

These extensions are necessary to support the database and applications models, as explained in the following sections.

2.3.2. The Database Model and Database-Independence

The database model must have <u>database-independence</u>, i.e., the database must be modeled independent of any real database. What characterizes a database?

First there are certain characteristics which reflect the physical dimensions of the database. These include the number of relations in the database, and the tuple width, the number of attributes per tuple, and the cardinality of each relation. Next there are characteristics which reflect the content of the database. These are the data types of the attribute values and the distributions of the attribute values. Finally, there are characteristics relating to access paths. These are the index structures and the distribution of data among devices in the database store.

We use a synthetic database for our benchmarks. The synthetic database is modeled independent of any real application. The number of relations in the database is fixed. The tuple widths, number of attributes per tuple, and cardinalities of the relations are based in part on the machine architecture. The data types and distributions for the attribute values which are important in the benchmarks are fixed. The index structures and distribution of data among the devices in the database store are machine-dependent. However, none of the parameters are chosen based on any real database. Therefore we maintain database-independence.

Relations are generated by a parameterized program, called a relation generator. Our model allows some flexibility. The format of each relation is based on a standard tuple template. To complete the design of the database model, parameters for some attribute values, tuple widths, and cardinalities of the relations must be supplied by the user.

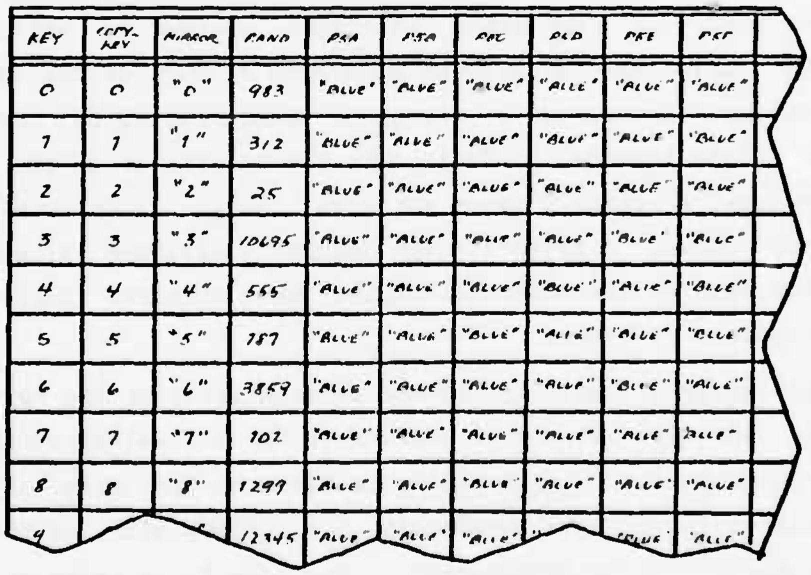
As shown in Figure 2, each tuple shall contain the following attribute values:

1. an integer "KEY" attribute value, generated sequentially;
2. an integer attribute "COPY_KEY", a copy of the integer key;
3. a character attribute value "MIRROR", which is the character representation of the "key" value;
4. an integer attribute value "RAND", generated randomly;

5. a character attribute value which represents a uniform distribution over 20 values, called "P5A"; the 20 values are the names of colors and the values are generated in blocks, i.e., the first 5% of the tuples have the identical values, the second 5% have the identical values, etc.

6. five additional attribute values, "P5B", "P5C", "P5D", "P5E", and "P5F" which are the same as "P5A".

The "KEY" attribute can be used as a primary key for the relation.



KEY	COPY-KEY	MIRROR	RAND	P5A	P5B	P5C	P5D	P5E	P5F
0	0	"0"	983	"BLUE"	"ALVE"	"ALUE"	"ALLE"	"ALUE"	"ALUE"
1	1	"1"	312	"BLUE"	"ALVE"	"ALUE"	"BLUE"	"ALUE"	"BLUE"
2	2	"2"	25	"BLUE"	"ALUE"	"ALUE"	"ALUE"	"ALUE"	"ALUE"
3	3	"3"	10695	"ALUE"	"ALUE"	"ALUE"	"ALUE"	"BLUE"	"ALUE"
4	4	"4"	555	"ALUE"	"ALUE"	"ALUE"	"BLUE"	"ALIE"	"BLUE"
5	5	"5"	787	"BLUE"	"ALUE"	"BLUE"	"ALIE"	"ALUE"	"BLUE"
6	6	"6"	3859	"ALUE"	"ALUE"	"ALUE"	"ALUE"	"BLUE"	"ALUE"
7	7	"7"	102	"ALUE"	"ALUE"	"ALUE"	"ALUE"	"ALLE"	"ALLE"
8	8	"8"	1299	"ALLE"	"ALUE"	"BLUE"	"ALUE"	"ALUE"	"ALUE"
9	[illegible]	[illegible]	12345	"ALUE"	"ALLE"	"ALLE"	[illegible]	"BLUE"	"ALLE"

Figure 2: The Standard Tuple Template

Using a primary key, we can impose an ordering on the tuples within a relation, and we can access directly any tuple of the relation. The "COPY_KEY" attribute allows us to compare performance for queries qualified on primary-key attributes with performance for queries qualified on secondary-key or unindexed attributes. The "MIRROR" attribute allows us to compare performance for integer vs. character key attributes. The "RAND" attribute can be used to reorder the tuples, so that the effects of ordering on performance can be studied. The "P5A" attribute gives us a vehicle for restricting the volume of data accessed from the disk in a predictable fashion. The "P5B", "P5C", "P5D", "P5E" and "P5F" attributes give us the capability to construct queries with successively increasing numbers of predicates while holding the volume of data accessed from the disk constant. Thus we can measure the effect of increasingly complex qualifications on performance.

The choice of additional attribute values to attain the required tuple widths is to be specified as part of the database model design. The number of attribute values should be the same for all tuple lengths, to facilitate interpretation of the benchmark results. Allowing 4 bytes for each integer attribute value specified above, 9 bytes for the "mirror" attribute value, and 6 bytes for each of the other attributes, the standard tuple template requires 51 bytes.

The choice of tuple widths and cardinalities of the relations is also variable, but should conform to the following recommendations. The choice of _tuple widths_ is based on the size of the essential unit of data management, i.e., block size, of the particular machine being benchmarked. The idea is to contrast measurements where there are many short tuples per block to those where there are a few long tuples per block. As the number of tuples per block increases, the work per block required of the processor increases. By varying the number of tuples per block, we can attempt to determine if/when the processor becomes the bottleneck.

We recommend that four tuple sizes be chosen: the first to yield a large number of tuples per block with a reasonable tuple length; the second a small number of tuples per block; the remaining two to represent some intermediate numbers of tuples per block. To facilitate interpretation of the results, the larger tuple widths should be chosen as multiples of the smallest tuple width. For example, if the unit of data management is a 4K-byte block, tuple widths of 100 bytes, 200 bytes, 400 bytes, and 2000 bytes is reasonable. This gives a range of 2 to 40 tuples per unit of data management.

We recommend three choices for the cardinalities of the relations. The idea is that, using identical queries against relations of increasing cardinality, we can identify the point at which I/O becomes a performance bottleneck. The first choice of cardinality should represent a small number of tuples, say 500. The next choice should represent a large number of tuples, and should explicitly satisfy the following criterion. If the machine stages data from the database store into memory, the volume of the largest relation, calculated as a product of the tuple width and the cardinality, should be at least twice as large as the total available memory. In the case where data is processed directly from the devices in the database store, the cardinality should be chosen such that the volume of the largest relation is at least twice as large as the volume of data which can be simultaneously accessed from the database store. The idea is to have at least one relation large enough to reveal any limitations imposed by the primary memory size or the bandwidth between the database store and the processors. The third choice should be an intermediate number between the largest and the smallest. For example, given the tuple widths in the preceding paragraph and a configuration with 2 million bytes of memory, the largest cardinality should be greater than or equal to 2000. 1000 is a logical intermediate choice. Again, the larger cardinalities are multiples of the smallest.

The relation generator is used to create a database of twelve relations, representing every combination of tuple width and relation cardinality. Is this arbitrary <u>number of relations</u> satisfactory? It is logical to assume that, in a relational database system, information about the database is kept in some set of system tables. One of the results shown in [Hawt79] is that references to system relations show a high degree of locality. It is reasonable to assume that access to these relations will be maximized either via some access path or by cacheing the data. Therefore, we argue that the number of relations in the database will not have a significant effect on performance.

<u>Data types</u> in our model are limited to integers and character strings. We have chosen not to include a floating point or decimal data type. It is not clear just how important arithmetic operations are in database management applications. Excluding them from our benchmarks is an arbitrary choice, and the deficiency can be easily remedied. Character string as well as integer types are included so that the relative efficiency of integer vs. character comparisons can be studied.

Next, we come to the question of modeling <u>data distributions</u>. It is our contention that the important factors in performance are the volume of relevant data and the volume of participating data. The <u>volume of relevant data</u> is the amount of data which must be accessed from the database store to process a query. The <u>volume of participating data</u> is the volume of data which participates in the answer to a query. The volume of relevant data determines the I/O workload. The volume of relevant data, together with the number of tuples per block, determines a component of processor workload. The additional component of processor workload is determined by the volume of participating data and the number of tuples per block (for example when an aggregate operation is specified in the query). Clearly the performance indices will vary as the ratio of these two factors varies. Rather than provide facilities to generate values according to different distributions, we provide the facility to generate values according to discrete distributions in multiples of 5%. We maintain that it is sufficient for benchmarking purposes to use only uniform distributions. The volume of relevant data and the volume of participating data can be accurately controlled and predicted by qualifying queries on attribute values with uniform distribution. The "p5" attribute value explained above is an example. By qualifying a query with the predicate

p5 = "value"

where "value" is one of the twenty values across which the attribute values were generated, we know that 5% of the tuples of the relation will participate in the answer. If we know that the attribute values were generated in contiguous blocks, and we know the tuple width, block size, and index structure, we can calculate the volume of relevant data. Although we maintain that uniform distributions are sufficient, the discrete distribution facility can be used to approximate other distributions.

Let us summarize our design decisions. We will use a database of twelve relations of synthetic data, generated by the relation generator. The twelve relations represent all possible combinations of four tuple widths and three relation cardinalities. All tuples will have the same number of attribute values. The tuple formats will be based on a standard tuple template, as described above. The attribute values will be of type integer or type character string. The attribute values will be integers generated in sequence, integers generated randomly, character strings in lexicographical order, or integers or character strings generated according to some discrete distribution.

2.3.3 The Applications Model and Application-Independence .

To achieve application-independence, the benchmark applications must be modeled independent of any real application. Our third design decision relates to the construction of the benchmark jobs. The basic units of our benchmark, the benchmark kernals, are the data management operations. The next higher level, the benchmark instructions, are queries.

The basic benchmark applications are modeled as collections of similar benchmark instructions. A variety of basic benchmarks are specified with the object of determining the essential performance characteristics of the machine. The basic benchmarks are thus independent of any real application. These benchmarks are run in single-user mode, giving us "best-case" performance measurements. The analysis of the performance statistics from these benchmarks will allow us to classify queries according to the scheme which is presented in Chapter 4.

The basic benchmarks are then divided into query pools on the basis of query classification. Multi-user benchmarks are then modeled as a collection of jobs. Each job is an instruction mix from a particular query pool, over a set of relations. By characterizing the workload with mixes of the basic benchmark instructions, we maintain application-independence. The performance measurements from these experiments are analyzed in relation to the best-case measurements of the single-user benchmarks.

3. A QUERY CLASSIFICATION SCHEME

Our applications model for multiple-user benchmarks is based on a classification of the basic benchmark queries. Several query classifications schemes have been proposed. In each case, the classification is made on an ad hoc basis. The rules for classification are imprecise. A scheme based on the queries categorizes queries as "simple" or "complex". Another scheme, based on the database rather than on the query, classifies queries as "low volume" or "high volume". A scheme which considers both the query and the database was used by Hawthorn and Stonebraker [Hawt79] in a performance analysis of INGRES. Queries were classified as data-intensive, overhead-intensive, and multi-relation. In Section 3.1, we will present a simple model of a backend database machine. In Section 3.2, we explain our query classification scheme as it relates to the database machine model. The method for classifying queries according to actual measurements is presented in Chapter 5.

3.1. A Simple Model of Query Execution

Let us construct a very simple model of a backend database machine. Let us assume 100% availability of resources, i.e., no waiting. Queries are parsed and formatted by the host computer, and transmitted to the DBM. The DBM first looks up the information needed to execute the query. This includes data from the system tables about relations, attributes, and indices, and the indices themselves. The DBM then executes the query, examining the relevant data block-by-block as it is read in from the database store.

We can develop formulae which represent the elapsed times for queries submitted through a host computer to a backend database machine. Since our intent is to motivate a query classification scheme, and not to build a useable analytic model, we make our first simplifying assumption: the elapsed time and its components are represented as <u>averages</u>. The average elapsed time for queries which involve no update can be stated as:

$$T = Th + Tc + Ts + i*Tx + (r + 1)*Td + b*(r + 1)*Tq + a*Tc$$

Each of the factors on the right hand side of the equation is discussed below.

Th is the average host overhead. Host overhead includes the time required to parse and format the query and the time required to deblock and process the tuples returned by the backend. If precompilation or some similar technique is available, the time required for parsing and formatting the query can be reduced to some small constant. We will assume that such a capability is available. The time required to deblock and process the tuples returned is clearly proportional to the number of blocks returned and the number of tuples per block.

Tc is the average time required to send one block across the host-backend communication link. We assume that blocks are fixed length, i.e., that short blocks are padded to the full block length.

Ts is the average time required in the DBM for initial setup. This includes the time required to access the data dictionary and to set up the parameters necessary for query execution.

i is the average number of blocks of index data to be accessed. Tx is the average time required to access and search one block of index data. i*Tx, then, represents the overhead incurred for an index.

(r + 1) is a variable which represents the number of blocks of <u>relevant</u> data, i.e., the number of blocks of the relation which must be read in from the database store in order to execute the query. Td is the time required to access one block from the database store. Td is assumed to be a constant, the average access time computed from the physical characteristics of the storage medium. (r + 1)*Td then, represents the average time required to access (r+1) blocks of data from the database store.

b is a variable which represents the number of tuples per block, and Tq is average time required for the processor to process one tuple. By "process" we mean to apply the operations specified in the query. b*(r + 1)*Tq, then, is an expression which represents the time required to execute the query.

a is a variable which represents the number of blocks of answer data, i.e., the number of blocks of data transferred from the backend to the host. The expression a*Tc, then, represents the total backend-to-host transfer time.

Now let us begin to simplify the expression. Let us make two additional simplifying assumptions. First, we will assume that transfer of data from the backend to the host is overlapped with deblocking and processing at the host. Second, we will assume that tuples can be deblocked and processed by the host at a rate which is greater than or equal to the data transfer rate. These two assumptions imply that the database machine never waits for the host to request a transfer of data. This is the ideal situation in that the performance of the database machine depends only on the resources of that machine, and not on the host machine resources. Performance measurements taken in this ideal situation represent the best-case performance of the database machine.

The effect of these assumptions on our formula is to reduce Th to some small, constant factor for parsing and formatting, Tp, plus the time required to deblock and process the final block transferred. According to our assumptions, the latter can be pessimistically estimated as the time required to transfer one block of data from the backend to the host, Tc. Therefore, we can substitute (Tp + Tc) for Th. So we have:

$$T = (Tp + Tc) + Tc + Ts + i*Tx + (r + 1)*Td + b*(r + 1)*Tq + a*Tc$$

Now let us assume that transfer of data to the host can be overlapped with accesses to the database store and query execution. This reduces the term which represents backend-to-host transfer time from a*Tc to Tc, representing the time required to transmit the last block. Let us also assume that accesses to the database store and query execution are overlapped. The time to access the database store and execute the query can be reduced to:

$$Td + \max(r*Td, b*r*Tq) + b*Tq.$$

Td represents the time to access the first block, and b*Tq represents the time required to process the final block. The expression max(r*Td, b*r*Tx) represents the remaining execution time. This indicates that execution time may be directly proportional to access time, in the case where queries can be executed at disk transfer rates. Otherwise, execution time will be determined by processing time.

As a final simplification, let us assume that index search is overlapped with access of blocks of index data. Since we expect that the index search algorithm will be fast, we can estimate the time required to process the index as i*Td, that is the time required to access the index.

We now simplify our formula for average elapsed time to:

$$T = Tp + 3*Tc + Ts + Td + i*Td + \max(r*Td, b*r*Tq) + b*Tq$$

Note that the first four terms are <u>constants</u>. These represent the fixed overhead for query. Let us combine them and write our formula as:

$$T = Ov + i*Td + \max(r*Td, b*r*Tq) + b*Tq,$$

where Ov represents the fixed overhead. The variables of the response time, as shown by the formula, are the time required to process the index, the time required to access the data from the database store, and the time required for processing the data.

3.2. The Three Query Classifications

In the previous section, we demonstrated that elapsed times are the sum of the time required to process the index, the time required to access the data from the database store, and the time required for processing the data. As demonstrated by the model, elapsed times are a function of the query variable i, the database variable b, the variable r, which relates the query to the database structure, and the constants which represent machine resources. The database machine resources are the processor or processors, and the memory, both secondary and primary. Consumption of processor resources is represented by the constant Tq. Consumption of memory resources is represented by the constant Td. There is some overhead in the host machine and in the database machine for every query, represented by the expression (Ov + i*Td).

We propose three query classifications which correspond to the components of the elapsed-time sum: overhead-intensive, access-intensive, and processor-intensive. Access-intensive queries are those for which the dominant factor in query execution time is the volume of relevant data accessed from the database store. Processor-intensive queries are those for which the dominant factor in query execution time is the work done by the processor. Overhead-intensive queries are those for which the dominant factor is the work done by the host and the database machine before accessing the relevant data from the database store, for example parsing and formatting the query or accessing and searching indices.

The model presented above is very simple and straightforward. For example, it does not consider the case that, for sorting or join operations, additional I/O operations may be required. However, the query classification scheme is applicable for all operations, since the elapsed time for any query can be characterized in terms of the query,the database, and the machine resources.

4. THE PERFORMANCE INDICES

A _performance_ _index_ is "a descriptor which is used to represent a system's performance or some of its aspects" [Ferr78]. When we evaluate a system or a machine, we are interested in both quantitative and qualitative aspects. The quantitative aspects are speed, productivity, and capacity. The qualitative aspects include such things as ease-of-use and the scope of functionality. For example, when we evaluate a relational database machine we might want to ask not only how fast the machine performs join operations, a quantitative aspect, but also whether the machine furnishes the capability to perform inequality joins, a qualitative aspect.

The primary object of this study is _quantitative_ performance evaluation. We have defined performance indices for the quantitative aspects of performance. Execution and interpretation of the benchmarks will provide some very useful qualitative information. However, we have not attempted to provide a complete evaluation of qualitative aspects. So, in the next three sections of this chapter, we define the _quantitative_ performance indices for comparative evaluation of relational database machines.

4.1. A Meaningful Performance Index

There are three classes of quantitative performance indices. Indices of _productivity_ measure the volume of information processed by the system per unit of time. Indices of _responsiveness_ measure the time between the presentation of an input to the system and the arrival of the corresponding output. Indices of _utilization_ measure the activity of a resource over an interval of time [Ferr78].

The database machine user is interested in all of these classes of performance indices. All of the following are questions a user might ask. What is the response time for query x? How many transactions of type y per minute can be processed? What percentage of the (CPU, I/O, memory, bus, communications) resources are absorbed by workload z?

4.2. What Measurement Tools are Available?

Two factors limit the choice of measurement tools. First, we are dealing with a variety of host machines and database machines. Some type of software or firmware monitor will usually be available for the host machine. A monitor may be available for the database machine. However, the monitors are designed for specific architectures. There is no simple way to specify measurements in such a way that the results from machine-to-machine will be comparable. Second, the database machine and related software are, in all probability, proprietary products. Modification of the software and the use of hardware probes within the machine will be prohibited. Even if these activities were not prohibited, deciding where to place the hardware probes and software traps would be a complex task, different for every machine.

One facility that is generally available on all systems is access through software to the system clock. Because of its simplicity and general availability, this is the performance measurement tool which we will use. Monitors and other tools may be used to provide additional information, but their use will not be addressed in this methodology. If such tools are used, care should be taken that adjustment be made for any overhead incurred by the tool.

4.3. The Measurements

The system clock will be our measurement tool. We will measure elapsed time. The time from the system clock will be recorded just prior to the submission of a query. The time from the system clock will be recorded again immediately after the last participating tuple has been returned from the database machine to the user process on the host.

Some overhead will be incurred in sampling the system clock. An estimate of this overhead can be derived by constructing a simple job which repeatedly samples the system clock. The job should be run under controlled conditions, with the host workload as specified in Chapter 2. The sample mean time between calls can be used as an estimate of the overhead. With a sample size of 30, we can assume an approximately normal distribution. The sample should be sufficiently large to establish a confidence interval for the sample mean with a confidence coefficient of at least 0.95.

4.4 The Performance Indices - Elapsed Time and Throughput

We will calculate two performance indices from our measurements. The first, the throughput index, measures the amount of work performed by a system in a given unit of time. This is an index of productivity. The second index, the elapsed time index, is defined as the interval of time required per unit of work. This is the reciprocal of the throughput index. The elapsed time index is an index of responsiveness. In Section 4.1, a third type of index was defined, i.e., indices of utilization. Since we have limited our measurement tool to the host system clock, we will not be collecting the type of data required to construct such an index.

The throughput and elapsed times indices are defined in terms of units of work. How do we define "work"? In the context of database systems, we have several choices. We might choose a query as a unit of work, or a transaction as a unit of work. Neither of these is a good choice, because the actual work involved in a query or a transaction is dependent not only on the form of the query or transaction, but also upon the amount of relevant data and participating data in the database. We choose instead to use the tuple as a unit of work. The exact definition of work is different for each query classification. These definitions will be given in the Section 4.4.1, where we define the performance indices for the basic benchmarks. The performance indices for the multiple-user benchmarks are defined in Section 4.4.2.

4.4.1. Performance Indices for the Basic Benchmarks

As the results of the basic benchmarks are analyzed, queries are assigned to query pools according to the query classification scheme presented in Chapter 3. Let us assume that as the queries are assigned to query pools, they are assigned unique identifiers. The access-intensive queries are assigned identifiers of the form A_i. The processor-intensive queries are assigned identifiers of the form P_j, and the overhead-intensive queries identifiers of the form O_k.

Now, let us define three functions of a query. Let x be a query identifier of the form A_i, P_j, or O_k. Then we define

E(x) to be the <u>elapsed</u> <u>time</u> for query x,

PT(x) to be the the number of <u>participating</u> <u>tuples</u> for query x, i.e., the number of tuples which satisfy the qualification of query x, and

RB(x) to be the number of <u>relevant</u> <u>blocks</u> of data for query x, i.e., the number of blocks of data accessed for query x.

Let us also define a constant, N_{max}, which is the largest number of tuples per block in the database.

We will calculate performance indices for each query pool. The formulae are given below.

$$(1) \qquad \overline{EA} = \frac{\sum_i E(A_i)}{(\sum_i RB(A_i)) \cdot N_{max}}$$

(2) $$\overline{EP} = \frac{\sum_j E(P_j)}{\sum_j PT(P_j)}$$

(3) $$\overline{EO} = \frac{\sum_k E(O_k)}{\sum_k PT(O_k)}$$

All of the indices are in units of time per tuple.

The mean elapsed time per tuple for access-intensive queries, $\overline{EA}$, is a quotient. The dividend is the sum of elapsed times for all access-intensive queries. The divisor is the sum of the number of blocks of relevant data for all access intensive queries multiplied by the maximum number of tuples per block in the database. The index represents the minimum elapsed time per tuple for access-intensive queries. The dominant factor in response times for these queries is the number of blocks of relevant data. Therefore, an index defined in units of elapsed time/block is the logical choice. However, since we want to state all indices in the same units, we use tuples per unit time.

The mean elapsed time per tuple for processor-intensive queries, $\overline{EP}$, is the quotient of the sum of elapsed times for all processor-intensive queries and the sum of the number of participating tuples for the same queries. This is a straightforward choice. The mean elapsed time per tuple for overhead-intensive queries, $\overline{EO}$, is the quotient of the sum of elapsed times for all overhead-intensive queries and the sum of the number of participating tuples for the same queries. We expect that overhead-intensive queries will be single-tuple queries, or queries involving a small number of tuples. It is reasonable to distribute the overhead incurred over the number of participating tuples for these types of queries.

What is the significance of these indices? In each case the elapsed time index represents the best-case performance of the database machine for a particular class of queries. The indices provide valuable information about the basic performance of the database machine. These indices will also be used in calculating comparative performance indices for the multiple-user benchmarks.

4.4.2 Performance Indices for the Multiple-User Benchmarks

Multiple-user benchmarks are modeled as collections of jobs. Each job is modeled as a collection of queries from a single query pool. The jobs, then, can be categorized as access-intensive, processor-intensive, or overhead-intensive. Assume a mix of 'j' jobs, of which 'a' jobs are access-intensive, 'p' jobs are processor-intensive, and 'o' jobs are overhead-intensive.

The first three performance indices are analogous to the mean elapsed time per tuple indices calculated for the basic benchmarks. The formulae are given below, where n indicates that these indices pertain to the nth multiple-user benchmark. Each index is the dividend of the mean elapsed time per job and the sum of the tuples for all jobs in a given category.

$$(4)\qquad \overline{EA}_n = \frac{\sum_{ajobs}\sum_{i} E(A_i)/a}{\left(\sum_{ajobs}\sum_{i} RB(A_i)\right) * N_{max}}$$

$$(5)\qquad EP_n = \frac{\sum_{pjobs}\sum_{j} E(P_j)/p}{\sum_{pjobs}\sum_{j} PT(P_j)}$$

$$(6)\qquad \overline{EO}_n = \frac{\sum_{ojobs}\sum_{k} E(O_k)/o}{\sum_{ojobs}\sum_{k} PT(O_k)}$$

Now we can construct an overall mean elapsed time per tuple index for this particular benchmark job mix. This index follows naturally from the elapsed time per tuple indices above.

$$(7)\qquad E_n = \frac{\sum_{jobs}\left[\sum_{i} E(A_i) + \sum_{j} E(P_j) + \sum_{k} E(O_k)\right]/j}{\sum_{jobs}\left[\left(\sum_{i} RB(A_i)\right) * N_{max} + \sum_{j} PT(P_j) + \sum_{k} PT(O_k)\right]}$$

This represents the mean elapsed time per tuple for the job mix.

The overall mean elapsed time per tuple index can be shown to be a function of the individual elapsed time per tuple indices as follows. Let x, y, and z represent the work component of the individual indices. Then we can write $\overline{E}_n$ as

$$(8)\quad \overline{E}_n = \frac{1}{\left[(1+\frac{y+z}{x})*n\right]}*\overline{EA} + \frac{1}{\left[(1+\frac{x+z}{y})*n\right]}*\overline{EP} + \frac{1}{\left[(1+\frac{x+y}{z})*n\right]}*\overline{EO}$$

To evaluate the effect of the job mix on performance, we construct comparative indices for each job classification and for the job mix. The measured elapsed times are replaced by an expression which contains the best-case elapsed times per tuple derived from the basic benchmarks. The indices for the job classifications represent the dividend of the mean best-case-estimated elapsed time per job and the total number of tuples for a job classification. The calculation is simplified to the mean best-case elapsed time over the number of jobs in the job classification.

$$(9)\quad \overline{EA}'_n = \frac{(\sum_{ajobs}\sum_{i} RB(A_i))*N_{max}*\overline{EA}/a}{\sum_{ajobs}\sum_{i} RB(A_i))*N_{max}} = \frac{\overline{EA}}{a}$$

$$(10)\quad \overline{EP}'_n = \frac{(\sum_{pjobs}\sum_{j} PT(P_j))*\overline{EP}/p}{\sum_{pjobs}\sum_{j} PT(P_j)} = \frac{\overline{EP}}{p}$$

$$(11)\quad \overline{EO}'_n = \frac{(\sum_{ojobs}\sum_{k} PT(O_k))*\overline{EO}/o}{\sum_{ojobs}\sum_{k} PT(P_k)} = \frac{\overline{EO}}{o}$$

The overall comparative index for the benchmark is the dividend of the mean elapsed time per job and the total tuples for the entire job mix.

$$(12) \quad \bar{E}'_n = \frac{\sum_{jobs}\left[(\sum_i RB(A_i)) * N_{max} * \overline{EA} + (\sum_j PT(P_j)) * \overline{EP} + (\sum_k PT(O_k)) * \overline{EO}\right] / j}{\sum_{jobs}\left[(\sum_i RB(A_i) * N_{max} + \sum_j PT(P_j) + \sum_k PT(O_k)\right]}$$

This represents the ideal machine performance, where the degree of multiprogramming has no effect on the performance of the machine. We can show that $\bar{E}'_n$ is a function of the individual comparative indices in the same manner as shown for $\bar{E}_n$. The ratio of $\bar{E}_n$ to $\bar{E}'_n$ represents the <u>relative efficiency</u> of the machine for this particular job mix and degree of multiprogramming.

We will also calculate mean throughput indices. The indices are calculated as follows, where 'c' represents some constant used to convert the time units. For example, if the elapsed times are measured in seconds/tuple, a constant of 60 is used to yield a throughput index in tuples/minute.

$$(13) \quad \bar{T}_n = \bar{E}_n^{-1} * c$$

$$(14) \quad \bar{T}'_n = \bar{E}'^{-1}_n * c$$

$\bar{T}_n$ is the actual mean throughput for this job mix. $\bar{T}'_n$ is the mean throughput calculated the for ideal machine. The definitions of the indices for the job classifications are similar.

Note that we have not specified the time units to be used. The time units of the raw measurements will depend on the system being benchmarked. However, we recommend that, for generality and comparability, the elapsed time indices be computed in seconds/tuple and the throughput indices in tuples/minute.

5. CONSTRUCTING AND EXECUTING BASIC BENCHMARK JOBS

Two parts of our methodology are specifications for benchmark job structures and a procedure for running benchmark jobs. In Section 5.1, we will describe the benchmark job structures. In Section 5.2, we will discuss the number of measurements to be taken per job. The actions which must be taken to prepare for executing the benchmarks are described in Section 5.3. The procedure for executing benchmark jobs is detailed in Section 5.4.

5.1. The Benchmark Job Structures

The basic benchmark jobs will be batch jobs. The object of the basic benchmarks is to establish optimum performance characteristics for the machine. Performance measurements for interactive jobs are limited by the data rate of the terminal line. Performance measurements for batch jobs are limited only by the amount of buffer space available in the host main memory. Therefore batch jobs will give better estimates for optimum elapsed times and throughput.

There are two batch job models, one for single-tuple queries and a second for multiple-tuple queries. A job for single-tuple queries will execute a collection of single-tuple queries against a single relation. A job for multiple-tuple queries will execute multiple queries against multiple relations of a single tuple size.

Production programs would execute some operations on the data returned from the database machine. We will not attempt to model the actions of particular kinds of production programs. Instead, the batch jobs will merely output the tuples returned from the database machine. The output should be directed to a spool file, to avoid the limitation of output device speeds. A desirable side-effect is that the results are then available for verification. The batch jobs will also output accumulated statistics to a statistics file, and will write the queries to a query file for constructing the query pools.

Figure 3 is the pseudo-code representation of the body of a batch program which executes single-tuple queries against a single relation. The lines of pseudo-code are numbered on the left. Lines 1 through 4 are intialization code. An array to hold the measurements, measurement_array, and an index to the array, query_counter, are cleared. Lines 5 and 16 represent some mechanism for constructing the stream of single-tuple queries. This mechanism will vary with different host interfaces. In general, it will require substitution of values in the qualification portion of the query. Measurements are accumulated in the array in lines 6 through 17. Lines 18 through 22 represent the output phase of the program. After all queries have been executed, the mean elapsed time per participating tuple, the variance, and the standard deviation are computed. An adjustment for the time required to read the host system clock should be subtracted from the mean calculated from the raw measurements. The raw measurements are then sorted into ascending order to facilitate analysis. The count of the number of queries, total elapsed time, the mean, variance, and standard deviation, as well as the raw measurements, are output to the statistics file.

Figure 4 is the pseudo-code representation of the body of a batch program which executes multiple-tuple queries against multiple relations having the same tuple size. The structure is similar to that of the single-tuple job shown in Figure 3. The major distinction is that we are dealing with multiple relations. A separate measurement array is kept for each relation (or set of relations in the case of multiple-relation queries), and statistics are calculated for each relation(s). The mechanism for constructing queries will require substitution of target relation

```
1.  begin single-tuple job;

2.   open output spool file,
        query file,
        statistics file,
        database;
3.   clear query_counter;
4.   clear measurement_array;

5.   construct first query;

6.   while (more queries) do;

7.      write query to query file;
8.      increment query_counter;
9.      begin_time = host system clock;
10.     submit query;
11.     get tuple;
12.     write tuple to spool file;
13.     end_time = host system clock;
14.     compute elapsed_time in seconds;
15.     measurement_array[query_counter] =
            elapsed_time;
16.     construct next query;

17.  end while;

18.  compute sum_of_elapsed_times,
          mean_elapsed_time_per_tuple,
          variance,
          standard_deviation;
19.  sort measurement_array into
     ascending sequence;

20.  write query-counter,
        sum_of_elapsed_times,
        mean_elapsed_time_per_query,
        variance,
        standard_deviation,
        measurement_array
        to statistics file;

21.  close files, database;

22. end single_tuple job;
```

Figure 3: Single-Tuple Job Model

```
1.  begin multiple_tuple job;

2.  open output spool file,
        query file,
        statistics file,
        database;
3.  clear query counters;
4.  clear measurement arrays;
5.  initialize array indices;

6.  construct first query;

7.  while (more queries) do;

8.     write query to query file;
9.     determine current relation(s);
10.    clear tuple_counter;
11.    increment query counter for
        current relation(s);
12.    begin_time = host system clock;
13.    submit query;

14.    while (more tuples) do;
15.       get tuple;
16.       write tuple to spool file;
17.       increment tuple_counter;
18.    end_while;


19.    end_time = host system clock;
20.    compute elapsed_time in seconds;
21.    store elapsed_time and
       tuple_counter in
       measurement array for
       current relation(s);

22.    construct next query;

23. end while;
```

Figure 4: Multiple-Tuple Job Model

```
24. for (each relation or set of relations) do;
25.     compute sum_of_elapsed_times,
        mean_elapsed_time_per_tuple,
        variance,
        standard_deviation,
        sum_of_relevant_blocks,
        mean_elapsed_time_per_block,
        variance,
        standard deviation;

26.     sort measurement array into
        ascending sequence by
        elapsed time;
27.     write relation identifier(s),
          query_counter,
          sum_of_elapsed_times
          mean_elapsed_time_per_tuple,
          variance,
          standard_deviation,
          sum_of_relevant_blocks,
          mean_elapsed_time_per_block,
          variance,
          standard deviation,
          measurement array
          to statistics file;

28. end for;

29. close files, database;

30. end multiple_tuple job;
```

Figure 4: Multiple-Tuple Job Model (Continued)

names as well as substitution of values in the qualification portion of the query. The mean elapsed time per participating tuple and the mean elapsed time per relevant block, along with variances and standard deviations, are computed from the measurement arrays.

5.2. How Many Measurements for Reliability?

The reliability of the benchmark measurements is related to the number of measurements. A single measurement cannot be considered to be reliable. We must have a number of measurements. How many measurements are required to insure that the mean elapsed time is a reliable estimate?

The measurements from a benchmark job represent a sample from a unknown distribution of elapsed times. What we are really asking then, when we ask how many measurements are required, is what is an adequate sample size? One way to answer this question is to use the Central Limit Theorem. The Central Limit Theorem can be interpreted as follows. If a large random sample is taken from any distribution with mean μ and variance σ^2, then the distribution of the random variable $n^{1/2}(X - \mu)/\sigma$ will be approximately a standard normal distribution [DeGr75].

First note that this applies only to random samples. The measurements taken for a particular benchmark are certainly not a random sample. However, the measurements for a single relation within a benchmark can be considered to be a random sample from the distribution of elapsed times for a particular type of query and a particular relation. We can apply the Central Limit Theorem at this level. With a sample size of 62, the probability that the sample mean is within 0.25 standard deviations of the true mean is 0.95. (See Appendix C for calculations.) With a sample size of 16, the probability that the sample mean is within 0.5 standard deviations of the true mean is 0.95. Given a sample size n, we can calculate the interval around the mean with a given probability. With a sample size of 30, the sample mean will be within 0.3578 standard deviations of the true mean with a probability of 0.95.

Now we have a means of determining the sample size and calculating the reliability of the measurements. Is this practical? We will be dealing with queries which return many tuples and queries which return few tuples. Consider a query for which the elapsed time measurement is 10 minutes. If we execute 62 such queries, the benchmark job will run for 10 hours. Clearly this is not practical.

Where the volume of data associated with the queries is low, we will base the sample sizes on calculations from the Central Limit Theorem. Where the volume of data associated with the queries is high, we will choose a reasonable sample size and judge the reliability of results in an intuitive manner. It is reasonable to expect that as the volume of data increases, the elapsed time measurements will increase. Therefore our first criterion for reliability is that the elapsed time be a monotonically increasing function of the number of participating tuples, the number of relevant blocks, or the number of operations executed by the processor(s). As a second criterion, we will require that results be reproducible. In general, we will repeat the jobs three times. The measurements from the three repetitions should be very close. As a further criterion, we expect the results to be consistent with the particular machine architecture. In Chapter 6, we discuss the interpretation of benchmark results relative to the architectural features of database machines.

5.3. Preparing to Execute the Benchmarks

The first step in preparing to execute the benchmarks is to define the set of benchmarks which have relevance to the particular machine being benchmarked. Appendix B is a list of basic benchmarks. Each is uniquely identified by a roman numeral. Some of the basic benchmarks, for example those dealing with comparison of alternative index structures, will not be applicable if the machine does not support those index structures. Select only the applicable benchmarks for the benchmark set.

The second step is to establish the artificial database. The choice of parameters for tuple width, relation cardinality, and number of attribute values per tuple are documented, along with the standard tuple templates. The relations are generated using the relation generator, and loaded into the database machine.

To completely describe the structure of the database, we must also specify the placement of the relations on the devices in the database store and the index structure. In order to simplify the discussion, let us develop some notation. Our database is comprised of twelve relations with four tuple widths and three cardinalities. Let us designate 'S', 'M', and 'L' to represent the small, medium, and large cardinalities, and let '1', '2', '3', and '4' represent the four tuple widths, from smallest to largest. We can then construct relation names which uniquely identify each relation. For example, relation S1 is the relation with the smallest tuple width and the smallest cardinality.

The placement of relations on the devices in the database store is an important issue in multiple-relation queries. In cases where the user can control the placement of relations, we will be interested in measuring the effect of alternative placement strategies. Initially, the entire database will be stored on one volume. During the course of the benchmarks, relation M1 will be moved to a second device. Experiments to determine whether this results in a performance improvement will be run. If a significant improvement results, relations M2, M3, and M4 will also be moved to the second device. If no significant improvement results, relation M1 will be restored to its original placement.

In the case that the machine supports alternative index structures, the index structure of the database should be constructed as follows. The primary index for each relation is to be built on the KEY attribute values. Secondary indices for each relation are to be constructed on the MIRROR and P5A attribute values.

The third step in preparing to execute the benchmarks is to develop program templates for single-tuple and multiple-tuple query jobs. Once this is accomplished, we are ready to begin construction and execution of the basic benchmark jobs.

5.4. A Procedure for Executing Basic Benchmark Jobs and Recording Measurements

The basic benchmark jobs are executed in single-user mode with no host machine workload other than that associated with using the database machine. To facilitate organization, the job names and file names should be informative. File names and job names for single-tuple query jobs include the benchmark identifier and relation name. File names and job names for multiple-tuple query jobs include the benchmark identifier and the tuple width. Using the notation developed in the previous section, the job names for Benchmark I might be I1JOB, I2JOB, I3JOB, and I4JOB. The job names for Benchmark V might be VS1, VS2, etc.

The benchmarks are executed in order by benchmark identifier. All jobs for a particular benchmark are executed consecutively. Results are accumulated job-by-job, Figure 5 shows a proposed form for accumulating statistics. Printouts of the related statistics files should be attached to this form. In addition to these statistics, the total run time for each job should be recorded. If the job is a multiple-tuple job, also record the number of queries in the job. These run times will be used to determine the run time for multiple-user benchmarks, as explained in Chapter 8.

After all the jobs of a benchmark have been executed, the results are analyzed. The general guidelines in Chapter 6 and any specific guidelines for the particular benchmark should be followed. The end result of the analysis is a classification of the queries. According to the classification, assign the queries to the proper query pool. The documents from this analysis and the accumulated statistics will be the documentation for the benchmark.

Benchmark ID: Benchmark Title:

General Form of the Query:

Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ/RB	

Figure 5: Proposed Form for Accumulating Statistics

6. INTERPRETING BASIC BENCHMARK RESULTS

The classification of basic benchmark queries depends upon interpretation of the benchmark results. In order to interpret these results, we must first analyze the architecture of the machine. This analysis of the machine architecture can be developed from published material, and does not require any special knowledge of a proprietary nature. Based on an analysis of the architecture, we can develop some expectations about the machine's performance. If we are to have confidence in the results of benchmarks involving a limited number of queries, we must analyze them in relation to these expectations.

The results of the basic benchmarks will be analyzed with respect to the expected performance, and some specific guidelines for interpretation. In section 6.1, we describe the features of database machine architectures which are most important to performance. Section 6.2 consists of a sample analysis of a particular database machine architecture. A set of specific guidelines for interpreting actual benchmark results is given in Section 6.3.

6.1. Architectural Features Affecting Performance

One of the steps in interpreting benchmark results is to develop an analysis of the machine architecture. It is important to note that this analysis can be based on reference manuals, promotional literature, and other published information. No proprietary knowledge is necessary. But this analysis of the architecture is intended to be useful for interpreting benchmark results. It must relate the features of the architecture to the performance. What features should we consider? In the remainder of this section, we examine a number of database machine architectures. We categorize the machines with respect to the architectural features which have an important effect on performance. We include the following database machines in our study: CASSM [Su79a, Su79b], RAP [Ozka75], DBC [Bann78c, Bann79], DIRECT [DeWi79], DBMAC [Miss83], MDBS [He83, Meno81], SABRE [Gard83], RDBM [Schw83], VERSO [Banc83], IDM [Brit83], and Tanaka's Data Stream Machine [Tana83]. Of these, only MDBS does not implement the entire set of relational operations. We include it, however, as an illustration of some important architectural principles.

We are still in the problem-solving stage of database machine research. We have moved from pencil-and-paper solutions into prototype implementation. However, there is as yet no clear consensus on the "best" architecture for a database machine. An attempt to develop a taxonomy of architectures which reflects performance characteristics is fruitless -- every architecture is unique! We can, however, categorize the designs in respect to several important issues.

The first of these issues is processor structure. Without exception, these database machines are based on some multiprocessor structure. A number of processors share the work, in most cases directed by a control processor. We must consider the role of the control processor and the roles of the other processors separately. We can also find some commonality in interconnection schemes between the processors and the database store. In Section 6.1.1, we discuss the role of the control processor. In Section 6.1.2, we categorize processor structure by processor function and interaction. A second issue is the interconnection of the processor and the database store. We categorize methods of interconnection in Section 6.1.3.

A third issue arises from alternative physical organizations. With two exceptions, the architectures which we will examine support the relational model with tuples stored contiguously. VERSO supports a variation on the universal relation scheme, with the concept of a V-relation. DBMAC uses a tuple-substitution technique with data pool storage. In Section 6.1.4, we discuss the performance implications of these physical organizations.

A final issue is the host-processor interconnection. The apparent performance of the database machine may actually be limited by the speed of the interconnection mechanism. This issue is examined in Section 6.1.5.

6.1.1. The Role of the Control Processor

With one exception, DBMAC, the database machines we will examine are organized with one control processor and one or more slave processors. The role of the control processor can be divided into two functional areas. First, the control processor performs administrative functions, such as communication with the host machine, distribution of work among the other processors, and management of system resources. Second, the control processor may be required to participate in query execution. The participation may be limited to interacting with the other processors during query execution, or may involve assuming some of the functionality of the database machine.

Because all communication with the host goes through the control processor, it is a potential bottleneck. A certain fixed overhead will be incurred in the control processor for every query. Distributing the work among the processors will be more or less complicated depending on the functional architecture of the machine. Where the queries follow a fixed path of execution with predetermined resource allocation, this work will be minimal. This is the case in MDBS. Where resources are allocated dynamically, as in DIRECT, more work will be done by the control processor.

The control processor will also participate in query processing. At a minimum, it will do query decomposition. Other tasks that may be required are parsing, query optimization, concurrency control, address translation, and index management. In extreme cases, the control processor may be the only processor that can perform complex operations such as join. Every increment in functionality increases the potential of the control processor as a bottleneck. The extent to which the control processor participates in query execution also has an effect on performance through message traffic. A significant share of the bus resources may be consumed by the exchange of messages between the control processor and the slave processors. This depends, in part, on the manner in which the processors are interconnected and on their functionality. We discuss both of these issues in the following sections.

6.1.2. Database Machine Processor Structures

Database machine processor structures can be categorized as homogeneous multiprocessor or heterogeneous multiprocessor organizations. _Homogeneous multiprocessor organizations_ are characterized by a number of processors with identical functionality. Inherent in these organizations is a high degree of _intra-query parallelism_. The execution of a single query can be distributed across multiple processors. _Heterogeneous multiprocessor organizations_ are characterized by a number of processors with specialized functionality. These organizations offer a high degree of _inter-query parallelism_. The functionally-specialized processors perform partial execution of different queries simultaneously. In the following sections, we examine some database machine architectures to illustrate our categorization.

A. Homogeneous Multiprocessor Architectures

RAP, CASSM, DIRECT, DBMAC, and MDBS are all example of homogeneous multiprocessor architectures. The first two, RAP and CASSM, are examples of _cellular architectures_. They are characterized by the presence of processors which process data directly out of the database store. A processor and its associated storage together comprise a cell. Figure 6 shows a cellular architecture.

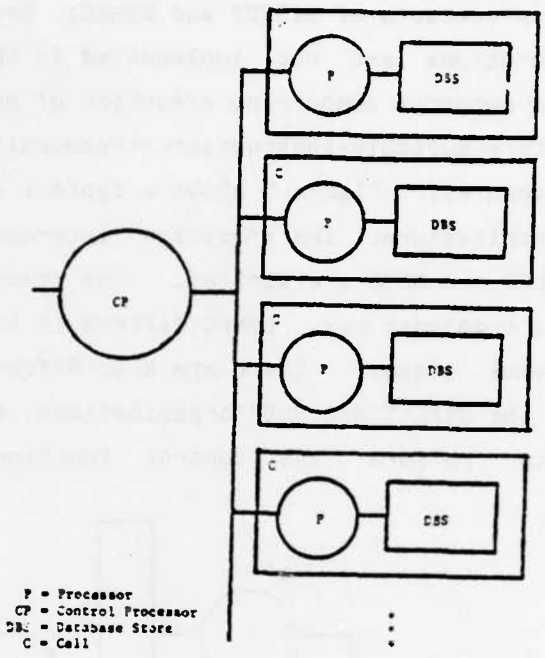


Figure 6: A Cellular Architecture

The processors are functionally simple, since they must operate at the rate of the device used for the database store. A query is broadcast to all processors, and each processor processes the query against the data available to it. There is no inter-query parallelism, only intra-query parallelism. These are single-instruction-stream-multiple-data-stream (SIMD) architectures [Flyn66]. The simple logic of the cell processors and absence of sophisticated processor interconnections are further limitations when operations involving two or more relations are processed. Such operations may have to be handled by the control processor or the host, or, as in the case of RAP, only a semijoin may be implemented.

In DIRECT, DBMAC, and MDBS, the processors are more sophisticated. The full range of relational database functions is implemented in the processors of DIRECT and DBMAC; however, the sort and join operations are not implemented in the MDBS. Each of these machines supports concurrent execution of multiple queries, and are therefore multiple-instruction-stream-multiple-data-stream (MIMD) architectures. Figure 7 shows a typical homogeneous multiprocessor architecture. The processor interconnection mechanisms of DIRECT and MDBS are similar. The processors are interconnected by a broadcast bus. DBMAC differs in that the processor bus is time-multiplexed. There are also differences in control strategy. In the DIRECT and MDBS organizations, one processor is designated to perform the control functions. Control is

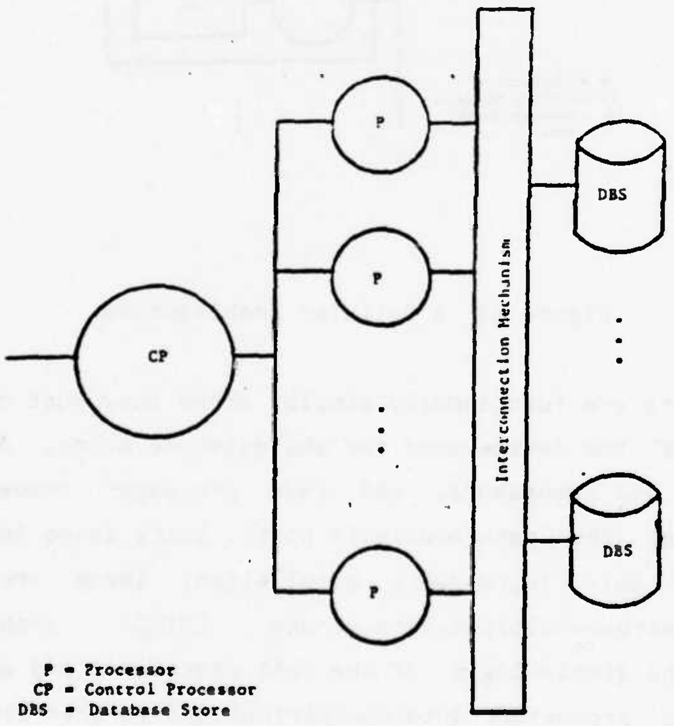


Figure 7: A Homogeneous Multiprocessor Architecture

distributed among the processors in the DBMAC organization.

In the DIRECT machine, the control processor assigns specific processors to specific queries. This strategy presents two problems. First, the control processor assumes the scheduling function, increasing its workload. Second, the individual processors work on only one query at a time. As a result, the processors will be idle while waiting for resources.

The scheduling function in MDBS is performed in the slave processors. The control processor broadcasts the queries to all slave processors. Each slave processor independently schedules queries for execution. The role of the control processor in query execution is minimized.

In the DBMAC machine, control is distributed among the individual processors. The unit of the work assigned to a processor is a "primitive" operation rather than a query. All transactions are translated into execution graphs, the nodes of which represent the primitive operations. This strategy results in less idle time in the processors. Distribution of the control among all processors will eliminate the control processor bottleneck problem. However, as a consequence of the distribution of control and the small unit of work assigned, message traffic is increased.

B. Heterogeneous Multiprocessor Architectures

DBC, IDM, SABRE, RDBM, VERSO, and Tanaka's Data Stream Machine are all examples of heterogeneous multiprocessor architectures. We can identify three sub-classifications. DBC is a pipelined architecture. IDM and VERSO are two-processor architectures. SABRE, RDBM, and the Data Stream Machine are dynamically configurable architectures.

Figure 8 shows the DBC architecture as an example of a pipelined architecture. All queries follow a fixed path through the pipeline. As is the case with all pipelined architectures, we can

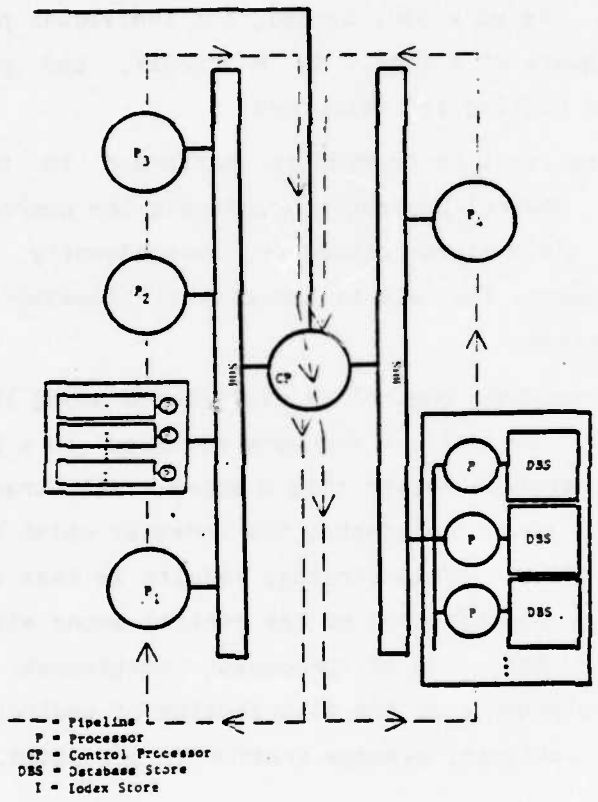


Figure 8: A Pipelined Architecture - DBC

expect the major performance advantage in throughput. The limiting factor is the speed of the processor in the pipeline which must do the most work. Pipelined architectures can be classified MIMD.

Figure 9 shows a typical two-processor architecture, consisting of a control processor and one slave processor. Each processor performs specific database functions. These are also

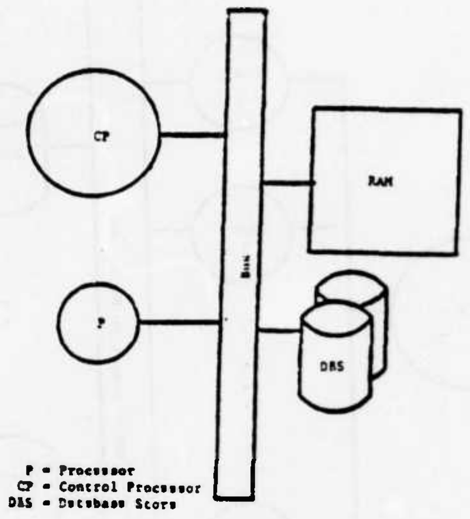


Figure 9: A Two-Processor Architecture

technically MIMD architectures. However, in the VERSO machine, the slave processor will participate in every query. In the IDM, the slave processor may or may not participate, depending on the operation. Therefore, the IDM might be pessimistically categorized single-instruction-stream-single-data-stream (SISD). The degree of inter-query parallelism is severely limited by the number of processors.

SABRE, RDBM, and Tanaka's Data Stream Machine are examples of dynamically configurable architectures. RDBM and Tanaka's machine dedicate separate processors for special functions. SABRE may assign several special functions to one processor. All are MIMD architectures. In all of these machines, a key factor is the algorithm used to allocate processor resources. Figure 10 shows a dynamically configurable architecture.

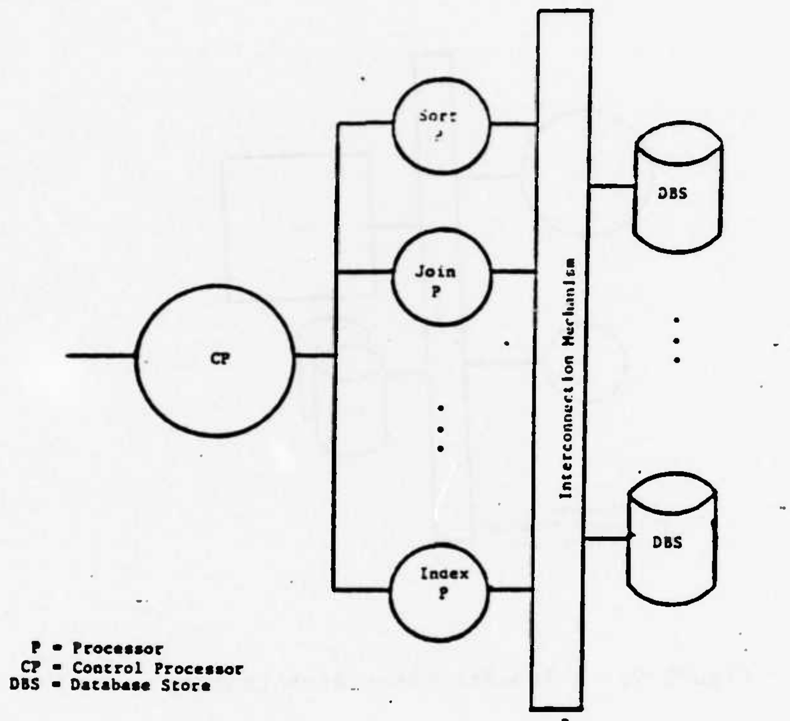


Figure 10: A Dynamically Configurable Architecture

6.1.3. Interconnection to the Database Store

The methods of interconnecting the processors and the database store can be divided into two major categories. First we have those architectures which associate a processor with every unit the database store. The cellular architectures, CASSM and RAP, are examples of this category. DBC also falls into this category. Let us call this category <u>direct interconnection</u>, to indicate that the processors are connected directly to the database store, with no staging of data. The second, a more common category, includes those architectures with some memory hierarchy. Data is staged from the database store into a RAM or RAMs, where it is then available for processing. Let us call this category <u>hierarchical interconnection</u>, to indicate that data is staged from secondary storage into some random access memory module or modules.

The direct interconnection strategy offers the advantage that processors never wait for data. The tradeoff is that the processors must be designed to work at the speeds which can keep up with the transfer rate of the secondary storage device. This may result in limited functionality, and increased cost for special storage technology. A second tradeoff is that the capability for sharing data among processors is severely limited. CASSM, RAP, and DBC use the direct interconnection strategy. Of the three, only RAP has a facility for inter-processor communication, and that facility is strictly limited to nearest-neighbor intercommunications for performing implicit joins. The interconnection strategy has been illustrated in Figure 6. Keep in mind, however, that this applies only to the MM (Mass Memory) processor in DBC.

Figure 11 shows a hierarchical interconnection scheme. Hierarchical interconnection strategies can be further subdivided according to two factors. First, there is the issue of which processors have access to the database store. Second, there is the issue of how data is shared among the processors once it has been staged into random access memory.

In the IDM, SABRE.V2, RDBM and VERSO machines, a single processor controls the transfer of data from secondary storage to random access memory. The IDM and SABRE.V2 machines transfer data across a single bus to the RAM. The performance of these machines is limited by the bus capacity, as shown in [Hsia83]. In the VERSO and RDBM machines, data from the disks is buffered. The data is then filtered out of the buffer before being sent to the random access memory, thus providing more efficient use of the bus. VERSO has a single filter processor; RDBM has multiple filter processors which operate in parallel.

The DIRECT, DBMAC, and MDBS machines provide parallel access to the database store. The DIRECT machine uses a crossbar switch

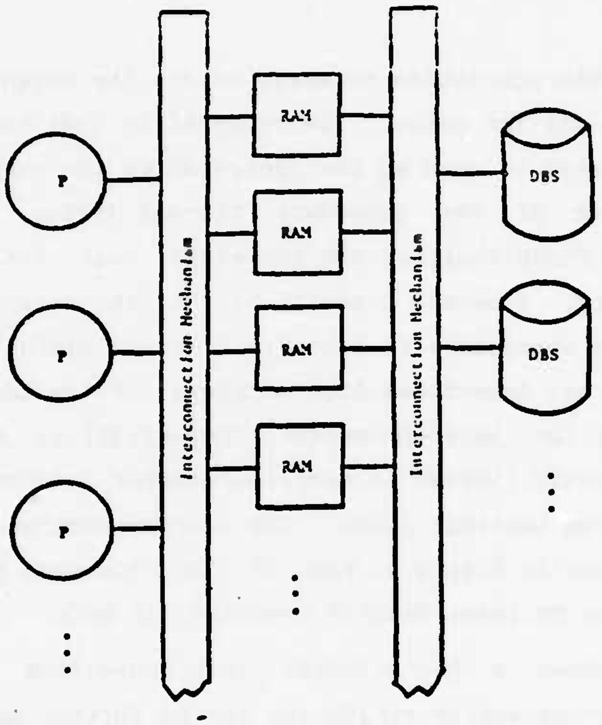


Figure 11: A Hierarchical Interconnection Scheme

to provide a virtual bus for each storage device-memory pair. This eliminates the single-bus limitation, but increases the workload of the control processor and the interconnection cost. The DBMAC machine has multiple special-purpose processors which filter data at disk transfer rates. The amount of data which must be brought in from secondary storage is thus reduced, allowing more effective use of bus capacity. In MDBS, every slave processor has access to a portion of the database on dedicated disks. The work is done in the slave processors, so that only participating data is placed on the bus.

6.1.4. Alternative Physical Organizations

DBMAC uses a data pool organization and a tuple substitution strategy. For a tuple accessed from the database store, the attribute values associated with the tuple id (TID) are selected from the data pools to compose the tuple. This has several implications. First, there is an additional overhead for tuple substitution in retrieving data from the database store. Second, projection operations may be more efficient, as only the attribute values specified in the target list will be accessed from the database store. Third, joins may be performed on their TIDs rather than on the tuples themselves [Blas77]. This may result in improved performance, depending on the actual volume of the TID information compared to the volume of the actual tuples. Fourth, insert and delete operations will be expensive, since separate accesses to each data pool may be required. Similarly, update operations may be expensive, depending on the number of attribute values updated.

VERSO uses a variation on the universal relation scheme, a V-relation [Banc83]. The V-relation has an attribute set and a set of update units. In an example from [Banc83],

R(course,student,grade,hour,room,teacher)

is a V-relation, with update units which correspond to the following set of normalized relations.

R1(course),
R2(course,student),
R3(course,student,grade),
R4(course,hour,room),
R5(course,teacher)

The update units represent logical relations. The tuples of the physical relations are of the form R((C(SG)*(HR)*T*)*). These tuples are stored in lexicographic order. This scheme has some advantages and some disadvantages. The main advantage is that joins are replaced by selections. However, updates are expensive due to the non-normal form of the physical relations.

To summarize, the motivation behind the use of data pools and universal relation schemes is to improve performance for some operations, for example join operations. This is accomplished only at the expense of other operations.

6.1.5. Host-Database Machine Interconnection

Let us examine a hypothetical database machine which can process data at disk transfer rates. Assume that the database is stored on conventional moving-head disks with a data transfer rate of about 1 Mbyte/sec. Let us process a hypothetical query on our hypothetical machine. Say that the query is a simple retrieval of all of the tuples in a relation, and that the volume of the relation is 1 Mbyte. Our elapsed time measurement should be somewhere in the neighborhood of 1 second _if_ the interconnection between the host and the database machine can support this transfer rate.

Two available interconnection devices are the RS-232 interface, with a data rate of 19,200 baud, about 2400 bytes/sec, and the IEEE-488 parallel interface, with a data rate of 170 K-bytes/sec. A third interconnection device is the selector channel, which may have data rates from 1 to 2 Mbytes per second [Baer80]. We must also consider devices such as Ethernet, with a data rate of .1 to 10 Mbits per second, or at maximum 1.25 Mbytes per second [Baer80]. Clearly only the selector channel or Ethernet can keep up with our hypothetical machine when the ratio of relevant data to participating data is 1:1. When the ratio is 5.88:1, the IEEE-488 parallel interface will then provide enough bandwidth. The RS-232 interface will be adequate only when the ratio of relevant data to participating data is as low as 404:1.

Clearly, the host-database machine interconnection will influence apparent performance. It may also limit the machine to less-than-capacity performance. Consider the case of our hypothetical machine. Say that we buffer the result data in RAM, and that the database machine-host transfer interface has DMA capability. The database machine can continue to process data even while data is being transferred to the host. If the host-database machine interconnection is slow, eventually, the situation will occur that the result buffers are full. The database machine must idle while waiting for buffer space.

What we must consider in this methodology is that the speed of the interconnection may influence our interpretation. A situation like that described above will cause some operations to appear to be access-intensive when they are actually overhead-intensive or processor-intensive. We can also expect that the relative efficiency measure will be somewhat overstated.

6.2. Expected Performance for a Database Machine

Let us choose a simple example. Figure 12 shows the machine architecture. The machine is organized around a single high-speed bus. There are two processors. The database processor, a

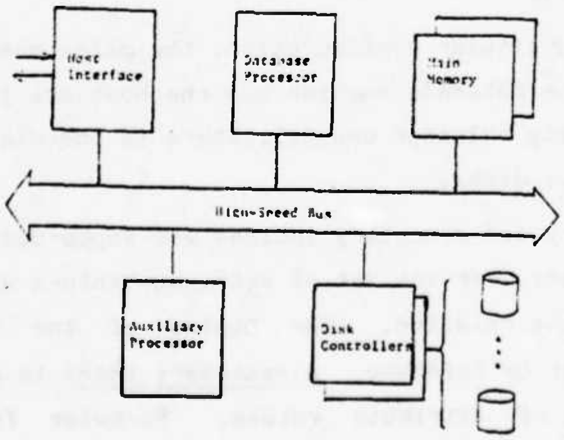


Figure 12: A Database Machine Architecture

conventional microprocessor, manages the machine resources as well as performing data management functions. The auxiliary processor, built from special-purpose hardware, performs a limited set of data management functions at high speed. The particular functions which it performs are unspecified. A host interface handles communication with the host. The intelligent disk controllers have the capability to do overlapped seeks. The database is stored on conventional, moving-head disks. The main memory is dynamic RAM.

Let us examine each of the features identified in the previous section. In this machine, the potential of the control processor to be a bottleneck is high, since it manages the machine resources and also participates in query execution. This is an example of a heterogeneous multiprocessor architecture. It is in fact a two-processor architecture, with a severely limited degree of inter-query parallelism. The interconnection between the processors and the database store is an example of hierarchical interconnection. The size of the primary memory is a potential limitation. Another potential limitation is that the machine has only a single bus.

In this particular configuration, the main memory capacity is 2 Mbytes. The database machine and the host are interconnected by a high-capacity selector channel. There is one disk controller for two 600 Mbyte disks.

Both primary and secondary indices are supported. A _primary index_ is an index over the set of attribute values which form the unique key for the relation. The tuples of the relation are stored in order by this key. A _secondary index_ is an index over an arbitrary set of attribute values. Formulae for computing index sizes are available.

As noted above, the degree of inter-query parallelism is severely limited. When the auxiliary processor, which can operate at disk transfer rates, is working, very little bus bandwidth will be available for the database processor. We know from the available literature that the functions of the auxiliary processor are limited. However, the exact functions are unspecified. It may perform the selection and projection functions and, perhaps, aggregation. It may not participate in join operations, since it is designed to operate on a single stream of data from the disks.

We expect that the performance for simple, high-volume, single-relation queries will in general be limited by disk transfer rules; the queries will be access-intensive. Low-volume, single-relation queries will probably be overhead-intensive. Where sorting is required, we expect that queries will be access-intensive. Multiple-relation queries such as joins will be either access-intensive or processor-intensive, depending on the join strategy used. If the strategy is based on sorting, the queries will be access-intensive. Other strategies will be processor-intensive or access-intensive, depending on the size of the relations relative to the size of the main memory. These are general expectations. Some operations or combinations of operations will not fit the general case. As these are revealed by the basic benchmarks, we must try to determine the cause and to classify the queries accordingly.

6.3. Guidelines for Interpreting Benchmark Results

The basic benchmarks are specified in the next chapter. The philosophy of these benchmarks is to execute similar queries on some subset of the relations in the database, and to analyze the measurements as the tuple width and the cardinality of the relations vary.

First let us review some definitions. As stated in Chapter 2, the important factors in performance are the volume of relevant data and the volume of participating data. The _volume of relevant data_ is the amount of data which must be accessed from the database store to process a query. The _volume of participating data_ is the volume of data which participates in the answer to a query.

We will usually express the volume of relevant data as a number of relevant blocks, and the volume of participating data as a number of participating tuples.

The measurements for each benchmark should be analyzed separately. Let us consider the analysis for multiple-tuple jobs and the analysis of single-tuple jobs separately.

6.3.1 Analysis of Measurements for Multiple-Tuple Jobs

The measurements for multiple-tuple jobs will be compared as tuple width and relation cardinality are varied. The queries are constructed such that when _tuple width varies and the cardinality is held constant_, the number of relevant blocks increases and the number of participating tuples remains constant. Graph the mean elapsed times per relevant block against the number of relevant blocks. If the mean elapsed times are a function of the number of blocks, the curve will be either flat or increasing. Where the curve is flat, the mean elapsed times are a constant function of the number of relevant blocks. An increasing curve may be showing either the effect of some bottleneck in the machine architecture, for example the limitation of the primary memory size, or the effect of some overhead operation, such as accessing and searching an index. A sharp increase generally suggests a limiting factor in the machine architecture. A more gradual increase generally suggests that some overhead is involved.

Where the curve is decreasing, we suspect that the queries may be processor-intensive, i.e., the mean elapsed times are a function of the number of participating tuples. In this case, graph the mean elapsed times per participating tuple, using tuple width as the unit on the x-axis. Tuple width is used on the x-axis since the number of participating tuples is the same for each query. If the queries are processor-intensive, the curve will be parallel to the x-axis.

Next we examine the case where _the cardinalities of the relations vary while the tuple width is held constant_. The queries are constructed such that the number of relevant blocks of data and the number of participating tuples increase in the same proportion as cardinality increases. We graph the mean elapsed time per participating tuple against the number of participating tuples. If the curve is parallel to the x-axis, the mean elapsed times are a function either of the number of relevant blocks or of the number of participating tuples. If this is the case, we will distinguish between the two by graphing the mean times per relevant block against the number of relevant blocks, and comparing those means to an an estimated mean access time per relevant block. The estimated mean access time is based on the measurements of Benchmark I. If these mean access times are greater than the estimated mean access time per block, then the queries are processor-intensive.

If, however, the mean elapsed time per participating tuple curve is increasing, the queries are either overhead-intensive or access-intensive. A sharp increase in the curve generally suggests the effect of some limiting architectural feature. A gradual increase generally suggests that the increase is due to overhead operations.

6.3.2. Analysis of Measurements of Single-Tuple Jobs

The single-tuple benchmark queries are inherently overhead-intensive. They are all single-tuple queries qualified on either primary or secondary index attributes. The single-tuple job model is a stream of like queries against a single relation. We expect that the elapsed time for the first query will reflect the overhead of accessing the index as well as searching the index. The elapsed times for the subsequent queries will reflect the overhead of searching the index. If the queries cause the index to be modified, there will be additional overhead for index modification, for logging, and perhaps for reorganization. In order to access a single tuple, a block of data will be read from the database store. Discounting the effect of cache hits, each measurement will include the time to access a block of data plus the overhead time.

We graph the mean elapsed times per participating tuple, using either tuple width or cardinality as the unit on the x-axis. We will use tuple width and cardinality since the number of participating tuples will not vary from job-to-job within a benchmark. With some machine architectures, the mean elapsed times, may actually reflect a mean elapsed time less that the estimated access time per relevant block for certain jobs. However, we will a priori classify all single-tuple queries as overhead-intensive or access-intensive, depending on the operation.

7. STANDARD BENCHMARK QUERY SETS

The core of the methodology is a collection of standard benchmark query sets. Each set consists of a number of experiments designed to measure the machine's performance for a particular operation with a variety of database structures. In the remaining sections of the chapter, we give specifications for the basic benchmarks.

For each benchmark, we will describe:

1) The form of the query;

2) The purpose of the benchmark;

3) The database structure and the set of relations against which the query is to be executed;

4) Recommended job structure and repetition factors;

5) Suggestions for interpreting results.

The form of the query will be specified in SQL [Cham74], the most widely-used relational query language. The notation is simple. Key words and attribute names are shown in capital letters. Substitutions must be made where elements are shown in lower case letters. For example, given the form

```
SELECT * FROM relation_name,
```

a valid relation name must be substituted at the position indicated by the "relation_name".

Recall from Chapter 5 that the number of measurements for low-volume queries is based on calculations using the Central

Limit Theorem (see Appendix C). The number of measurements for high-volume queries is chosen in an intuitive manner. A reasonable number of measurements will be suggested. Jobs for high-volume queries are repeated to determine reproducibility of the measurements. The measurements should be consistent as well as reproducible.

7.1. I - Unindexed Selection of 5% of Tuples

The form of the query is

```
SELECT * FROM relation_name WHERE P5B="value".
```

Relation_name is the name of one of the relations in the database, and "value" is one of the 20 of P5B.

The purpose of this benchmark is to establish baseline measurements for the best-case performance of the machine for access-intensive queries. We expect that the performance will be directly proportional to the number of relevant blocks, i.e., the number of blocks of data accessed from the database store. In this case, the number of relevant blocks is equal to the number of blocks in the relation. Since the attribute on which the query is qualified is unindexed, the entire relation will be examined.

This benchmark will be run against all relations in the database. Construct four multiple-tuple jobs. The first job will consist of queries against relations S1, M1, and L1. The second job will consist of queries against relations S2, M2, and L2. The third job will consist of queries against relations S3, M3, and L3. The fourth job will consist of queries against relations S4, M4, and L4. Each job will consist of three queries, one against each relation. The values for the qualification should be chosen randomly from the set of twenty values of P5B.

Run the four jobs in order three times. Since the volume (tuple width * cardinality) of relation L4 exceeds the memory capacity, this strategy eliminates any possibility of cache hits.

For each relation, list the number of relevant blocks and the total elapsed time. Compute the mean elapsed time per relevant block for each relation. Plot the mean elapsed times using number of relevant blocks as the unit on the x-axis. Then connect the points for relations having the same tuple widths. We expect the mean elapsed times to be a constant function of the number of relevant blocks. This means that the lines will overlap. If this is the case, the graph will be a single line parallel to the x-axis, as shown in Figure 13. We are making a very strong assumption that this will indeed be the case. In interpreting results of further benchmarks, we will use the results of this benchmark to estimate access time per relevant block for comparison purposes. If indeed the mean elapsed times are not a constant function of the number of relevant blocks of data, we have revealed a severe performance deficiency.

7.2. II - Selection of 5% of Tuples Qualified on Secondary-Key Attribute

The form of the query is

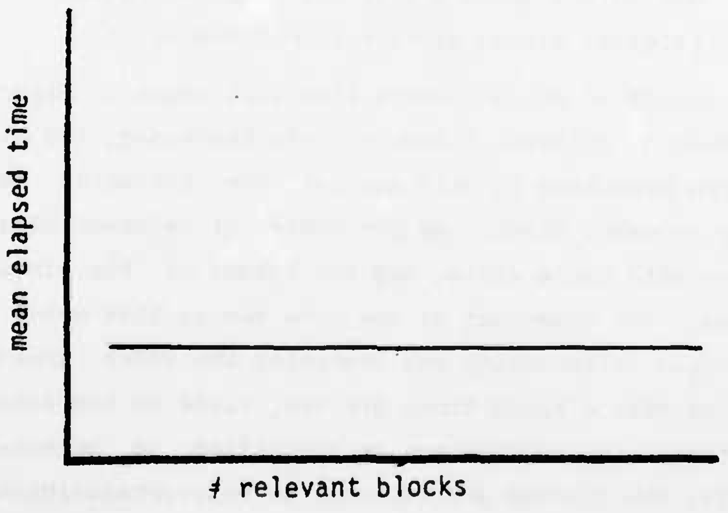


Figure 13: Benchmark I - Elapsed Times a Constant Function of # Relevant Blocks

SELECT * FROM relation_name WHERE P5A="value".

The purpose of the benchmark is to establish baseline measurements for selections qualified on secondary index attributes. The number of relevant blocks of data will be approximately 5% of the total number of blocks of data in the relation. This follows from the fact that the P5A attribute values were generated such that the first 5% of the tuples have identical values for P5A, the second 5% of the tuples have identical values for P5A, etc. A secondary index for the P5A attribute values exists.

This benchmark will be run against all relations in the database. Four jobs will be required. The first will consist of queries against relations S1, M1, and L1, the second of queries against relations S2, M2, and L2, etc. Each job will consist of three queries, one against each relation.

Run the four jobs in order three times. Compute the mean elapsed time per relevant block for each relation. Plot these mean elapsed times, using the number of relevant blocks as the unit on x-axis. Connect the points for relations having the same tuple widths. Add to the graph a line for the estimated mean access time per relevant block, derived from Benchmark I.

We expect to see a results like that shown in Figure 14. As the number of relevant blocks of data increases, the mean elapsed times from Benchmark II will approach the estimated mean access time per relevant block. As the number of relevant blocks of data increases with tuple width, and the volume of the index remains the same, the component of the mean access time which represents the overhead of accessing and searching the index grows smaller. Where the mean elapsed times are very close to the estimated mean access times, the queries can be classified as access-intensive. Otherwise, the queries are expected to be overhead-intensive.

To see the improvement in mean elapsed times with a secondary index, use the mean elapsed times per participating tuple from

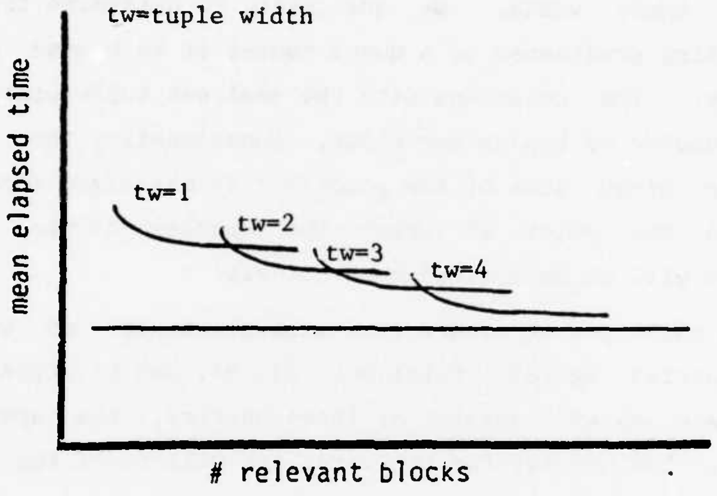


Figure 14: Benchmark II - Mean Elapsed Times Approach Estimated Access Time

this benchmark and from Benchmark I. For each relation, calculate the quotient of the mean elapsed time per participating tuple from Benchmark I and the mean elapsed time per participating tuple from Benchmark II. This is the improvement factor. Plot the quotients, using cardinality as the unit on the x-axis. Connect the points which represent queries against relations with the same tuple width. The improvement factor has an absolute upper bound of 20. If no overhead is incurred, the time to access 5% or 1/20th of the tuples will be 1/20th of the time required to access all of the tuples of a relation.

7.3. III - Increasingly Qualified Selection with ANDed Predicates

The form of the query is

```
SELECT * FROM relation_name
     WHERE P5A="value" AND ...
```

The purpose is to add predicates to the qualification to determine the point at which the queries become processor-intensive.

This benchmark will be run against the relations having the smallest tuple width. We are trying to determine the point at which adding predicates to a query causes it to become processor-intensive. The relations with the smallest tuple width have the largest number of tuples per block. Consequently, the amount of work per block done by the processor is maximized for our database, and the point at which the queries become processor-intensive will be determined more quickly.

The basic job structure is a multiple-tuple job which executes queries against relations S1, M1, and L1. Construct five jobs. Each job will consist of three queries, one against each relation. The queries for the first job will be of the form

```
SELECT * FROM relation_name WHERE
     P5A="value" AND P5B="value".
```

The queries for the second job will be of the form

```
SELECT * FROM relation_name WHERE
     P5A="value AND P5B="value" AND P5C="value".
```

Continue in this manner, adding predicates for attributes P5D, P5E, and P5F. The fifth job will consist of queries with six predicates in the qualification.

If the machine has primary memory, and the volume of relation L1 (tuple width * cardinality) is less than the primary memory capacity, flush the memory between runs to eliminate cache hits. Plot the mean elapsed times per participating tuple for relations S1, M1, and L1 from Benchmark II, using number of participating tuples as the unit on the x-axis. Connect the plotted points and label the curve to indicate a single predicate. These represent queries with a single predicate. Run the first job. Plot the mean elapsed times per participating tuple, connect the points, and label the curve to indicate two predicates. Run the second job and plot the measurements in the same manner. Continue with the third, fourth, and fifth jobs, until a clear trend emerges or all jobs have been executed. Then repeat the jobs twice to take two additional sets of measurements. reconstructing the graph to reflect mean elapsed time including results from both runs.

We may see results something like those in Figure 15. At some number of predicates, the mean elapsed time measurements will increase significantly, indicating the increase in the workload of the processor. These queries would be processor-intensive. Where no significant increase in mean elapsed time measurements is detected, the queries would be, like those from Benchmark II, overhead-intensive.

7.4. IV - Increasingly Qualified Selection With ORed Predicates

The form of the query is

```
SELECT * FROM relation_name WHERE
     P5A="value" AND
     ( P5B="value" OR P5C="value" OR ...)
```

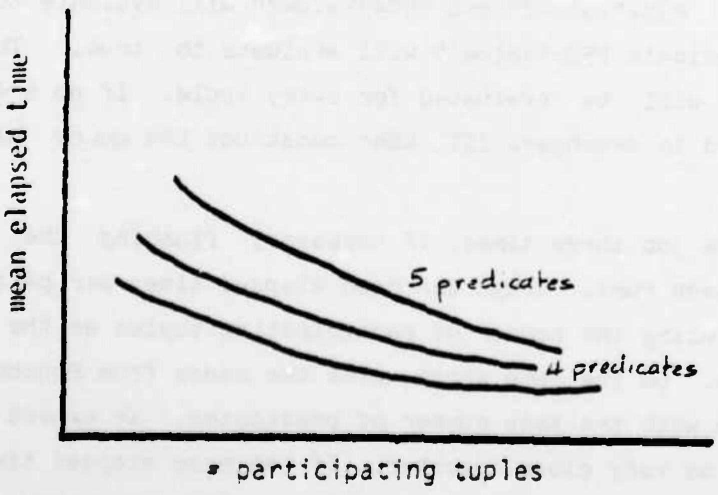


Figure 15: Benchmark III - ANDing Predicates to Determine Processor-Intensive Breakpoint

This is a limited experiment to determine whether queries with ORed predicates can be processed in a single pass through the data.

This benchmark will be run against the relations having the smallest tuple width, for the same reasons as Benchmark III. The basic job structure is a multiple-tuple job which executes queries against relations S1, M1, and L1. Construct a job with queries of the form given above. The job will consist of three queries, one against each relation. The total number of predicates should be chosen based on the breakpoint found in Benchmark III. For example, if Benchmark III shows that the queries become processor-intensive when the qualification has four predicates, the queries for Benchmark IV will be of the form

```
SELECT * FROM relation_name WHERE
        P5A="value1" AND
        (P5B="value2" OR P5C="value2" OR P5D="value1")
```

Given that the values of P5A, P5B, P5C, and P5D are identical, the predicates P5B="value2" and P5C="value2" will evaluate to false, and the predicate P5D="value1" will evaluate to true. Thus all predicates will be evaluated for every tuple. If no breakpoint was detected in Benchmark III, then construct the query with six predicates.

Run the job three times, if necessary flushing the primary memory between runs. Graph the mean elapsed times per participating tuple, using the number of participating tuples as the unit on the x-axis. On the same graph, plot the means from Benchmark III for queries with the same number of predicates. We expect the two curves to be very close together. If the mean elapsed times vary significantly, one possible explanation is that more than one pass through the relation is required when processing ORed predicates. If this is the case, the queries will be either processor-intensive or access-intensive.

7.5. V - Single-Tuple Selections Qualified on Integer, Primary-Key Attribute

The form of the query is

SELECT * FROM relation_name WHERE KEY=value.

The purpose is to measure the mean elapsed time for a sequence of single-tuple queries as cardinality and tuple width vary.

This benchmark will be run against relations L1, L2, L3, L4, S4, and M4. The measurements for the first four relations will show the effect of increasing tuple width on mean elapsed times. As tuple width increases and cardinality is held constant, the size of the index remains the same and the probability of cache hits drops. The measurements for the last three relations listed will show the effect of increasing cardinality on mean elapsed times. As cardinality increases, the size of the index increases, and the probability of cache hits drops. The relations having the largest tuple width are used to minimize the cache hits.

The basic job structure is a single-tuple job. Six jobs will be required. Each job will execute sixty-two queries against one of the relations. The measurements from each job will show the effect of cache hits if the machine has primary memory. The values for the qualification are selected randomly from a range of values from 0 to (cardinality-1). Use a different seed for the random numbers for each job.

Run the six jobs in order three times. Graph the mean elapsed time per participating tuple from the first four jobs, using tuple width as the unit on the x-axis. Graph the mean elapsed time per participating tuple from the last three jobs, using cardinality as the unit on the x-axis. The mean elapsed times are plotted against tuple width and cardinality, since the number of participating tuples will be 62 in each case and the number of relevant blocks will be approximately 62. Add a line to each graph which represents the estimated mean access time per relevant block derived from Benchmark I. These are single-tuple queries, so the access of a single tuple corresponds to the access of one block from the database store.

These queries are by definition overhead-intensive. We expect the curve on the first graph to be relatively flat, since the cardinality and the index volume remain constant. However, we expect the curve on the second graph to rise, as index volume increases with cardinality. The distance between the two lines on each graph represents the overhead of accessing and searching the index. It is conceivable that, due to cache hits, the measurements may in some cases be less than the estimated mean access time per relevant block.

7.6. VI - Single-Tuple Selections Qualified on Character, Primary-Key Attribute

The form of the query is

```
SELECT * FROM relation_name WHERE MIRROR="value".
```

This is an optional benchmark to determine whether, given a choice, character or integer keys are more efficient.

This benchmark will be run against relations S4, M4, and L4. As cardinality increases, the index size increases. Thus, the benchmark will provide a comparison of character vs. integer keys as the index size increases.

We must modify the database structure for this benchmark. Delete all the indices for relations S4, M4, and L4. Build a primary index on the MIRROR attribute values. Build secondary indices on the KEY and P5A attributes.

Three single-tuple jobs are required, one each for relations S4, M4, and L4. The appropriate jobs from Benchmark V can be modified for this purpose. The only change is that the qualification will be on the MIRROR attribute, which requires a character string value rather than an integer value.

Run the three jobs in order. Then rerun the jobs. Superimpose on the second graph from Benchmark V the mean elapsed times per tuple from this benchmark. If there is a significant difference in the curves, then the curve with the lower mean elapsed times indicates the most efficient index, an index character keys or an index on integer keys.

This benchmark is included for comparative purposes only. The queries will not be assigned to any query pool. When the benchmark is complete, restore the database to its previous structure.

7.7. VII - Unqualified Selection with Ordering on Non-Key Attribute

The form of the query is

```
SELECT * FROM relation_name ORDER BY COPY_KEY
```

The purpose of the benchmark is to measure the performance of the machine for sorting operations as cardinality and tuple width vary.

This benchmark will be run against relations S1, M1, L1, L2, L3, and L4. Comparing the measurements for the first three relations listed above will show the effect of increasing cardinality on mean elapsed times. As cardinality increases and tuple width is held constant, the processor does the same amount of work per relevant block while the number of relevant blocks increases. Comparing the measurements from the last four relations will show the effect of increasing tuple width on mean elapsed times. As tuple width increases and cardinality is held constant, the processor does the same amount of work for each query, but less work for each relevant block, while the number of relevant blocks increases.

Four multiple-tuple jobs are required. The first will consist of one query each against relations S1, M1, and L1. The second job will execute one query against relation L2, the third one query against relation L3, and the fourth one query against relation L4.

Run the four jobs in order three times. Graph the mean elapsed times per relevant block for relations S1, M1, and L1, using number of relevant blocks as the unit on the x-axis. Add to the graph a line representing the estimated mean access time per relevant block. We expect to see results like those shown in Figure 16. Where the curve is parallel to the x-axis, the queries are processor-intensive. The same amount of work is required per block regardless of the cardinality of the relation. This may be the case when the tuples can be sorted in the memory. Where the curve is rising, indicating that mean elapsed time increases with the number of relevant blocks, the queries are access-intensive. Graph the mean elapsed times per relevant block for relations L1, L2, L3, and L4, using number of relevant blocks as the unit on the x-axis. Add to the graph a line representing the estimated mean access time. We expect to see a graph similar to that shown in

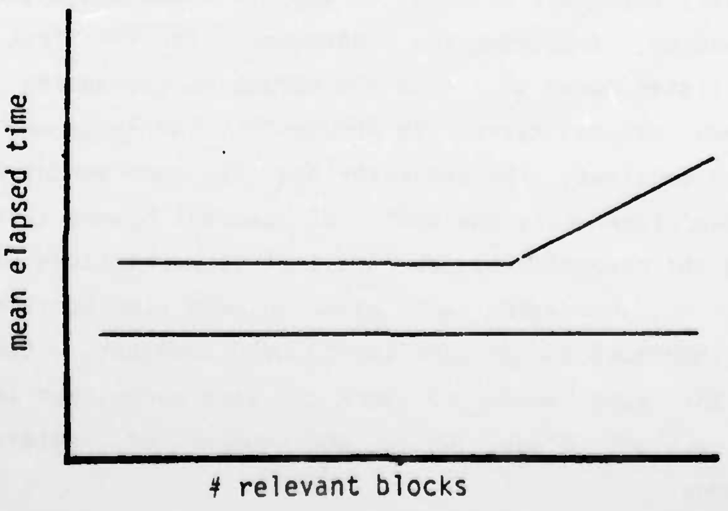
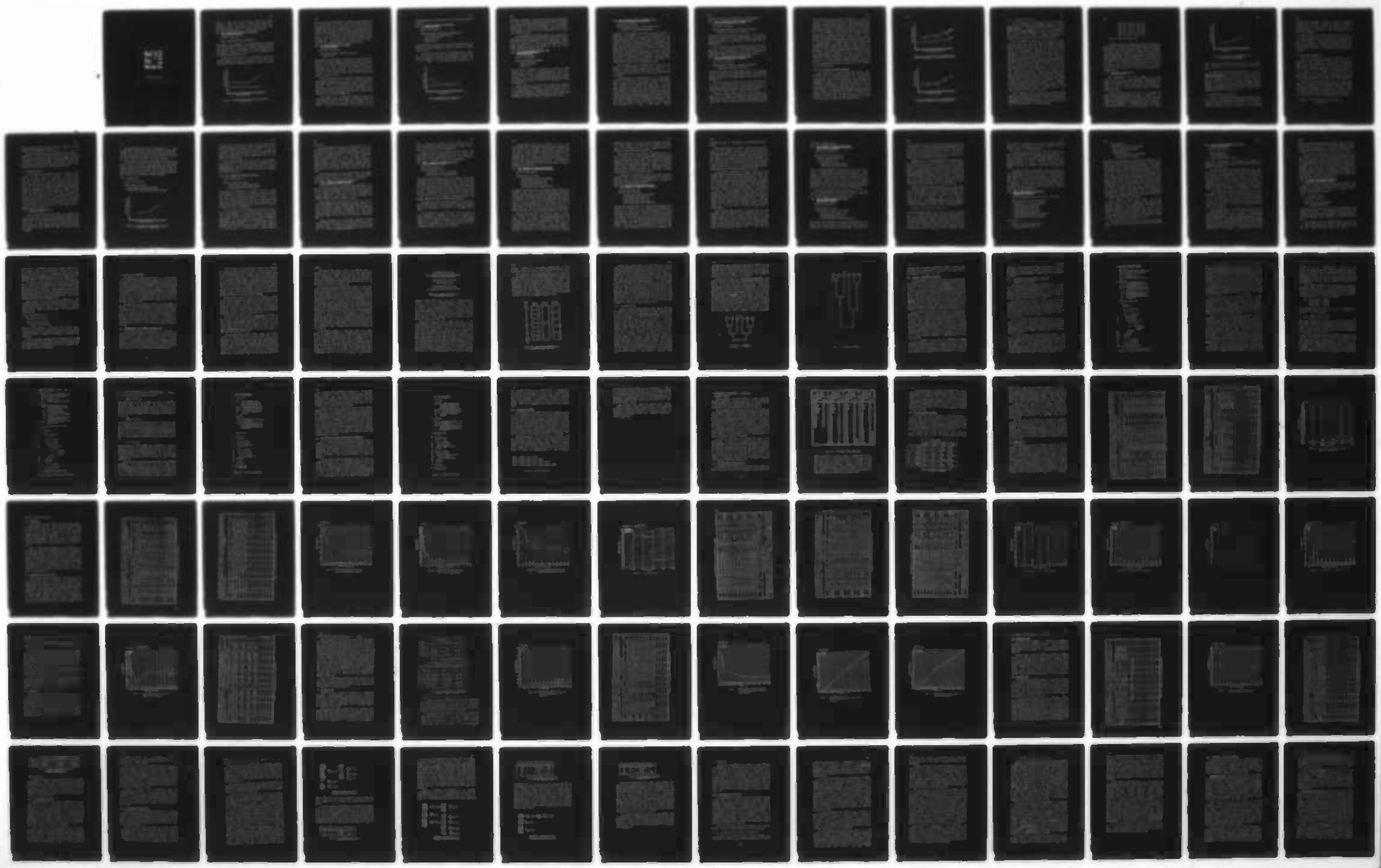


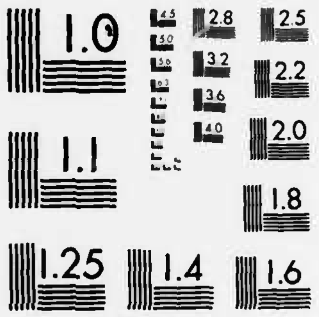
Figure 16: Benchmark VII - The Effect of Increasing Cardinality

AD-A139 104 A METHODOLOGY FOR BENCHMARKING RELATIONAL DATABASE MACHINES(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P R STRAWSER JAN 84 NPS52-84-004

2/3

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Figure 17. Where the curve is rising, queries are access-intensive. Where the curve is falling, the queries are processor-intensive. Less work per block is required when there are fewer tuples per block, i.e., as tuple width increases.

7.8. VIII - Unqualified Selection with Ordering on Primary Key Attribute

The form of the query is

SELECT * FROM relation_name ORDER BY KEY.

The purpose of this benchmark is to determine whether the machine uses the knowledge that the tuples are stored in order by primary key when asked to sort on the primary key. This benchmark will be run against relations S4, M4, and L4, to give an adequate basis for comparison with Benchmark VII.

Construct one multiple-tuple job, to execute one query each against relations S1, M1, and L1. Run the job three times, flushing the primary memory between repetitions if required. On a copy

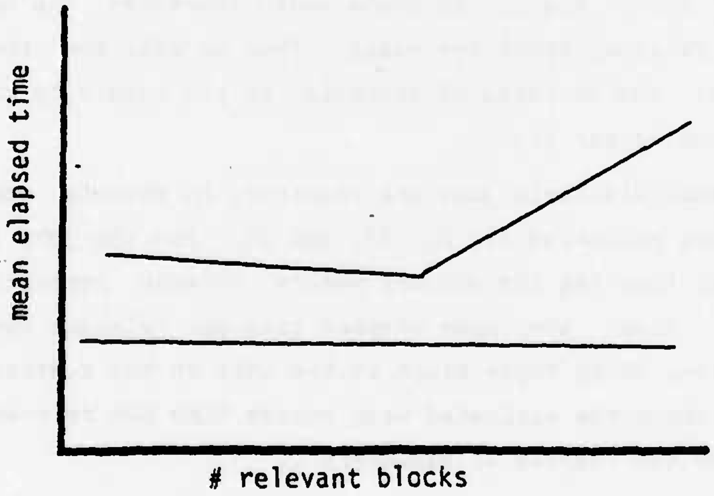


Figure 17: Benchmark VII - The Effect of Increasing Tuple Width

of the first graph from Benchmark VII, graph the mean elapsed time per relevant block for each relation. If the machine uses the knowledge that the tuples are ordered by primary key, the mean elapsed times for this benchmark should be significantly lower than those from Benchmark VII. If the mean elapsed times are very close to the estimated mean access time, this confirms that the machine is using the knowledge that the tuples are stored in order by primary key. If this is the case, the queries will be classified either as overhead-intensive or as access-intensive.

7.9. IX - Single Aggregate with Grouping on Non-Key Attribute

The form of the query is

```
SELECT COUNT( COPY_KEY ) FROM relation_name
     GROUP BY P5B.
```

The purpose of this benchmark is to measure the performance of the machine for collecting summary information by groups. This benchmark will be run against relations with the same cardinality and increasing tuple width. As tuple width increases, the number of tuples per relevant block decreases. Thus we will see the relationship of the overhead of accumulating the counts by group and number of tuples per block.

Four multiple-tuple jobs are required, to execute one query each against relations S1, S2, S3, and S4. Run the jobs in order three times, flushing the primary memory between repetitions if necessary. Graph the mean elapsed time per relevant block for each relation, using tuple width as the unit on the x-axis. Add a line which shows the estimated mean access time per relevant block derived from the results of Benchmark I.

These queries are by definition processor-intensive. We expect the graph to look like that shown in Figure 18. The effective overhead decreases as the tuple width increases and the number of tuples per block decreases. The difference in the two lines represents the overhead required for accumulating the counts by group.

7.10. X - Multiple Aggregates with Grouping on Non-Key Attribute

The form of the query is

```
SELECT MIN(COPY_KEY),MAX(COPY_KEY),SUM(COPY_KEY),
    COUNT(COPY_KEY),AVERAGE(COPY_KEY)
    FROM relation_name
    GROUP BY P5B.
```

The purpose of this benchmark is to measure the performance of the machine when multiple aggregate operators are applied to a single attribute value.

The relations and job structure are the same as the relations and job structure for Benchmark IX. Superimpose on a copy of the graph from Benchmark IX the mean elapsed times per relevant block for each relation. We may find that the mean elapsed times for

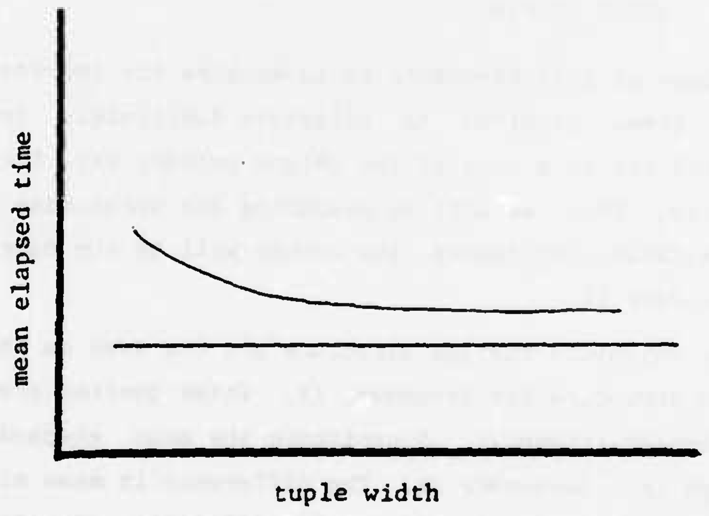


Figure 18: Benchmark IX - Overhead for Grouping on Non-Key Attribute

Benchmark X are greater than the mean elapsed times for Benchmark IX. A small increase may result from the additional work required for the processor to accumulate five statistics for each group, rather than one statistic for each group. Where the curve for this benchmark is close to the curve for Benchmark IX, the queries are processor-intensive.

If the accumulation of multiple statistics on a single attribute value requires multiple passes through the relation, then the mean elapsed times for Benchmark X will be significantly greater than for Benchmark IX. The curve for this benchmark will turn upward relative to the curve for Benchmark IX. Then the queries may be classified as access-intensive.

7.11. XI - Single Aggregate with Duplicates Eliminated and Grouping on Non-Key Attribute

The form of the query is

```
SELECT COUNT(DISTINCT COPY_KEY)
          FROM relation_name
          GROUP_BY P5B.
```

The purpose of this benchmark is to measure the increase in mean elapsed times required to eliminate duplicates. In this case, since COPY_KEY is a copy of the unique primary key, there are no duplicates. Thus we will be measuring the worst-case performance for eliminating duplicates. The counts will be the same as those for Benchmark IX.

The relations and job structure are the same as the relations and job structure for Benchmark IX. These queries are by definition overhead-intensive. Superimpose the mean elapsed times on the graph from Benchmark IX. The difference in mean elapsed times represents the overhead required for eliminating duplicates.

7.12. XII - Projection of 25% of Attribute Values Qualified on Secondary Key Attribute

The form of the query is

```
SELECT DISTINCT (COPY_KEY  other_attribute_names)
     FROM relation_name WHERE P5A="value"
```

The purpose of the benchmark is to measure the work involved in projection. The target list is a list of attribute names. The number of attribute names in the target list is 25% of the number of attribute values in a tuple of the relation relation_name. For example, if the tuples of relation S1 have 24 attribute values, then COPY_KEY and 5 additional attribute names will appear in the target list when relation_name is S1. There is one additional restriction. The primary key should not be included in the target list. The primary key is by definition unique. If the primary key is included in the target list, no overhead for eliminating duplicates will be incurred. Including COPY_KEY in the target list guarantees the worst case for eliminating duplicates.

Four multiple-tuple jobs are required. The first job will consist of queries against relations with tuple width 1, the second job of queries against relations with tuple width 2, etc. A total of three queries per job, one against each relation, will be included.

Run the four jobs in order three times. Plot the mean elapsed times per relevant block, using number of relevant blocks as the x-axis. Connect the points and mark the line "25%", to indicate 25% projection. Then construct four graphs, one for each tuple width, graphing mean elapsed time per participating tuple and using number of participating tuples as the unit on the x-axis. On each of these four graphs, also graph the mean elapsed time per participating tuple from the measurements of Benchmark II, Selection of 5% of Tuples Qualified on Secondary Index Attribute. Mark the lines to indicate 0% and 25% projection. Results from the next two benchmarks will be added to these graphs. Postpone analysis until the next two benchmarks have been completed.

7.13. XIII - Projection of 50% of Attribute Values Qualified on Secondary Key Attribute

The form of the query is

```
SELECT DISTINCT (COPY_KEY   other_attribute_names)
     FROM relation_name where P5A="value"
```

The target list should include the names of 50% of the attribute values in a tuple of the relation relation_name. The target list should be constructed in the same manner as the target list for Benchmark XII.

The job structure is the same as the job structure for Benchmark XII. Jobs should be run in the same manner also. Add the measurements from this benchmark to the graphs constructed in Benchmark II, marking the lines to indicate 50% projection.

7.14. XIV - Projection of 75% of Attribute Values Qualified on Secondary Key Attribute

The form of the query is

```
SELECT DISTINCT (COPY_KEY   other_attribute_names)
     FROM relation_name where P5A="value"
```

The target list should include the names of 75% of the attribute values in a tuple of the relation relation_name. The target list should be constructed in the same manner as the target list for Benchmark XII.

The job structure is the same as the job structure for Benchmark XII. Jobs should be run in the same manner also. Add the measurements from this benchmark to the graphs constructed in Benchmark II, marking the lines to indicate 75% projection.

Now we are ready to analyze the results from Benchmarks XII, XIII, and XIV. Examine one-by-one the graphs of mean elapsed times per participating tuple constructed separately for each tuple width. It may be the case that some mean elapsed times are lower than the mean elapsed times for 0% projection. This may be a result of a reduction in the volume of relevant data in machines which filter data on-the-fly from disk. Where measurements for projection are greater than those for 0% projection, the increase may represent either the increased work being done by the processor or I/O overhead. Use the graph of mean elapsed times per relevant block to determine which is the major factor. If the related mean elapsed time per relevant block curve turns up sharply past a given point, this generally indicates that access time is the major factor. For example, Figure 19 shows the case where mean elapsed time per participating tuple for 75% projection is greater than mean elapsed time per participating tuple for 0% projection. Three points on the 75% curve are identified as means for queries against relations L1, L2, and L3. Figure 20 shows mean elapsed time per relevant block for 75% projections. The points L1, L2, and L3 correspond to the points marked on the curve in Figure 19. We conclude that the queries against relations L1 and L2 are processor-intensive, and the queries against relation L3 are access-intensive.

7.15. XV - Equijoin on Non-Key Attribute

This is the first of a series of benchmarks involving joins. The sets of joins for subsequent benchmarks derive from the selection of the set of joins for this benchmark. The guidelines for interpretation given for this benchmark also apply for subsequent benchmarks. In Section 7.15.1, the method for selecting the set of joins for this benchmark is described. Section 7.15.2 contains the guidelines for interpreting measurements for joins as cardinality varies. The guidelines for interpreting measurement for joins as tuple width varies are set forth in Section 7.15.3. Finally, in Section 7.15.4, we describe Benchmark XV, giving the form of the query, the purpose, and the job structure.

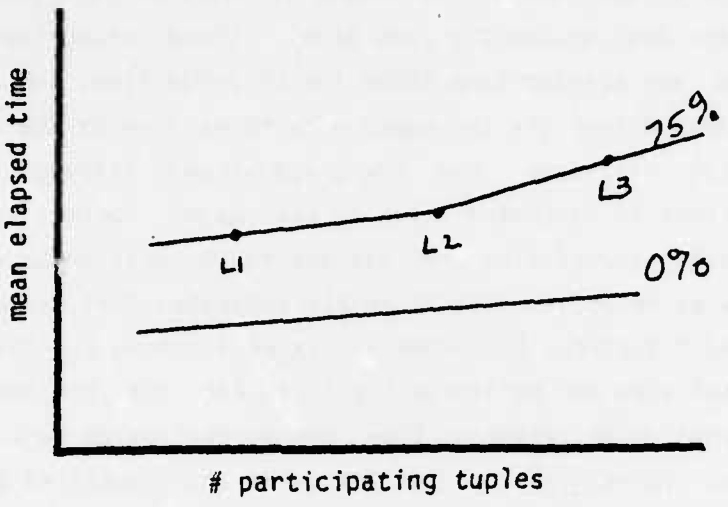


Figure 19: Benchmark XIV - Mean Elapsed Time Per Participating Tuple For 0% and 75% Projection

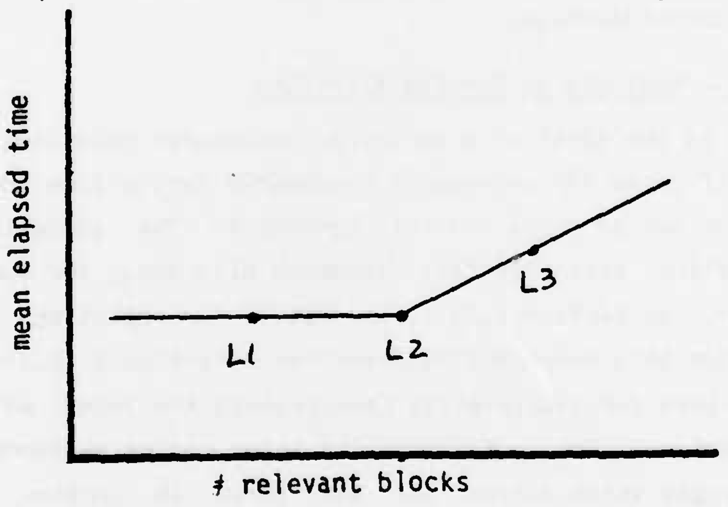


Figure 20: Benchmark XIV - Mean Elapsed Time Per Relevant Block For 0% and 75% Projection

7.15.1. Choosing the Set of Joins

First let us define some terminology. The two relations being joined are referred to as the source relation, S, and the target relation, T. The tuples which are returned to the user comprise the result relation, R. Let C(x) be a function which, given a relation name as an argument, returns the cardinality of the relation, i.e., the number of tuples in the relation. We establish two requirements for the joins in our benchmarks.

The first requirement is that the join be made over attributes for which the values are unique. The joins will be made over the KEY attribute values or over the COPY_KEY attribute values. We know that the values of *KEY and COPY_KEY are* unique integers in the range [0,C(x)]. Therefore, we know that, when performing a join over all the tuples of relations S and T, the cardinality of the result relation, R, will be min{ C(S),C(T) }. The second requirement is that the relations being joined have the same tuple width, and that, except where expressly stated otherwise, all of the attributes of a tuple of the source relation will be concatenated with all of the attributes of a tuple from the target relation when forming a tuple of the result relation. This will facilitate interpretation. Because the relations being joined have the same tuple width, we also know that the number of blocks of data in the result relation is exactly twice the number of blocks in the relation with the lesser cardinality. Let B(x) be a function which, given the name of a relation as an argument, returns the number of tuples per block for that relation. Then we can write the expression for number of relevant blocks in the result relation as 2 * min{ C(S),C(T) }/B(S).

Table 1 shows the matrix of possible joins. We would like to measure the performance of the machine for equijoins while varying tuple width and cardinality separately. The set of joins for this benchmark will include the joins specified in the first row of the table to measure the effect of varying cardinality.

S1 X M1	S1 X L1	M1 X L1
S2 X M2	S2 X L2	M2 X L2
S3 X M3	S3 X L3	M3 X L3
S4 X M4	S4 X L4	M4 X L4

Table 1: Matrix of Possible Joins

To choose the joins for measuring the effect of varying tuple width, examine the table row-by-row, marking those join specifications where the volume of the larger relation equals or exceeds the primary memory capacity or the volume of data that can be accessed simultaneously from the database store. Now examine the table column-by-column. Select the first column in which two join specifications have been marked. Add those two joins to the set of joins. If the marked specifications are in the third and fourth rows, also add the join from the second row of the column to the set of joins. Thus our set of joins will contain either six or seven joins.

7.15.2. Analysis of Measurements as Cardinality Varies

The measurements for the subset of joins {S1 X M1, S1 X L1, M1 X L1} show the effect of varying the cardinality of the relation while the tuple width is held constant. We expect that where the tuple width is small, and the number of tuples per block is large, the queries will be basically processor-intensive.

To show the work done by the processor, we will graph the mean elapsed time per operation, as shown in Figure 21. The number of operations required will depend on the join algorithm. We will propose guidelines for analysis with respect to three different join algorithms; a straightforward join algorithm, a sort-and-match algorithm, and a hashing algorithm. It is impor-

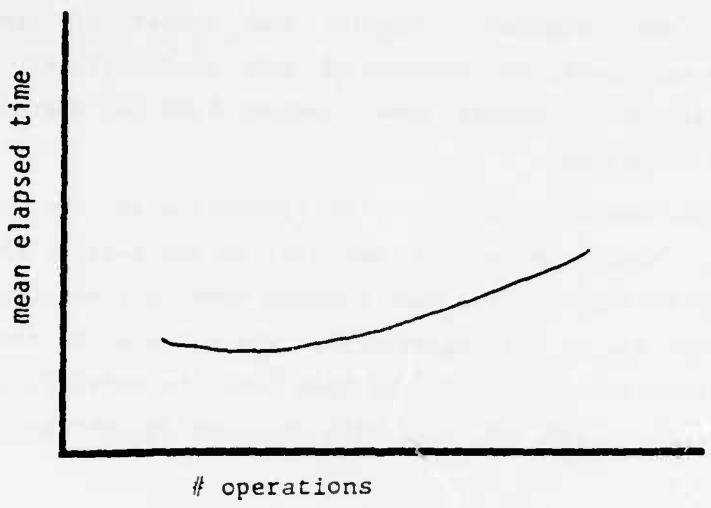


Figure 21: Benchmark XV - Mean Elapsed Time Per Operation for Joins of Relations with Small Tuple Width

tant to note that the machine may use more than one join algorithm. Therefore more than one set of guidelines may be applicable.

The means elapsed times from this benchmark will be used as baseline measurements for the joins S1 X M1, S1 X L1, and M1 X L1. The baseline measurements will be used in the analysis of measurements of subsequent benchmarks. The only exception is that, if Benchmark XVII shows that an alternative placement strategy results in a significant performance improvement, then the measurements from Benchmark XVII will be used as the baseline measurements for the joins S1 X M1 and M1 X L1.

A. A Straightforward Join Algorithm

First let us assume that a straightforward algorithm is used. Every tuple in the target relation is compared to every tuple in the source relation. This type of algorithm is also called a serial join [Chri81], a nested loop join [Seli79], or a join using tuple substitution [Wong76]. A total of $C(S)*C(T)$ comparison operations are required. Compute the number of operations involved in each join, the product of the cardinalities of the joined relations. Compute mean elapsed time per operation for each of the relations.

Graph the means, using number of operations as the unit on the x-axis. Where the curve is parallel to the x-axis, the query is processor-intensive. The mean elapsed time is a constant function of the number of operations. Where the curve rises, the query is access-intensive. In the case that the curve falls, some more efficient algorithm is being used and further analysis is required.

B. A Sort-and-Match Algorithm

Now let us assume that a sort-and-match algorithm is used. This type of join algorithm is also called a parallel join [Yao78], or a merge join [Seli79]. Assuming that a comparison-based sorting algorithm is used, the time required to sort n tuples will be $O(n \log n)$, where log n is the base 2 logarithm of n. Then, the time required for sorting will be $O(C(S)*(\log C(S)) + C(T)*(\log c(T)))$. The matching in this case will require $O(\min\{C(S),C(T)\})$ time. The total time required, then, is $O(C(S)*(\log C(S)) + C(T)*(\log C(T)) + \min\{C(S),C(T)\})$.

Let OPS(S,T) be a function which, given as arguments the names of the source and target relations, returns the the number of operations required when a sort-and-match strategy is used. Let $f(y)=[y]$ be the greatest integer function. Recall that $C(x)$ is a function which, given a relation name, returns the cardinality of the relation. Then we can define OPS(S,T) as follows.

$$\begin{aligned} OPS(S,T) = f(&C(S)*(\log C(S)) + C(T)*(\log C(T)) \\ &+ \min\{C(S),C(T)\}) \end{aligned}$$

Using the elapsed times per join and OPS(S,T), compute the mean elapsed time per operation for each join.

Graph the mean elapsed times per operation, using number of operations as the x-axis. Where the curve is parallel to the x-axis, the join is processor-intensive. The mean elapsed times are a constant function of the number of operations. Where the curve rises, the join is access-intensive. If the curve falls, then some other, more-efficient join algorithm is being used.

C. A Hashing Algorithm

A third possible join algorithm is based on hashing. When hashing is used, the time required is O(C(S) + C(T)). C(S) + C(T), the total number of tuples in the source and target relations, represents the number of operations required for the join. Using the elapsed times per join, compute the mean elapsed time per operation, the quotient of elapsed time and number of operations. Graph the mean elapsed times per operation, using number of operations as the unit on the x-axis. The interpretation is similar to those given above. If the results do not seem to fit any one of these algorithms, consider whether more than one algorithm is being used. It may also be necessary to consider other algorithms. The method of calculating the number of operations determined in this benchmark will be the method used in Benchmarks XVII through XXII.

7.15.3. Analysis of Measurements as Tuple Width Varies

We have selected joins where the volume of the larger relation is greater than or equal to either the memory capacity or the volume of data which can be accessed simultaneously from the database store. In this subset of joins, the tuple width is varied and cardinality is held constant. The work required of the processor remains the same, while the number of relevant blocks increases.

We expect these queries to be access-intensive, i.e., mean elapsed times increase as the number of relevant blocks increases. Therefore we will graph the mean elapsed times per relevant block, as shown in Figure 22. The number of relevant blocks for a join is the sum of the number of blocks in the joined relations.

Graph the mean elapsed times, using number of relevant blocks as the unit on the x-axis. Note that since the cardinalities remains constant, the number of comparisons required, and thus the amount of work done by the processor, remains constant. Where the curve is parallel to the x-axis, the query is processor-intensive. Where the curve rises, the query is access-intensive.

7.15.4. Benchmark XV

The form of the query is

```
SELECT (target list) FROM
    relation_name1, relation_name2 WHERE
    relation_name1.COPY_KEY=relation_name2.COPY_KEY
```

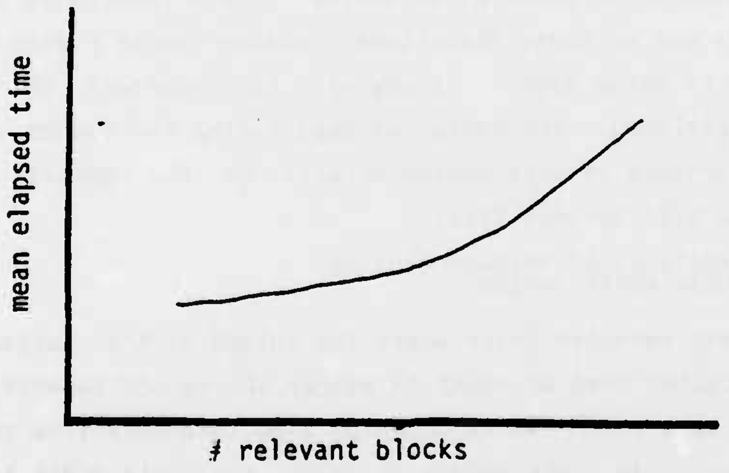


Figure 22: Benchmark XV - Mean Elapsed Time Per Block for High-Volume Joins

The purpose of the benchmark is to measure the worst-case performance of the machine for equality joins. The target list is a list of the names of all attributes of both relations. In the join predicate, relation_name1.COPY_KEY = relation_name2.COPY_KEY, the name of the relation with the smaller cardinality should be substituted for relation_name1.

Construct multiple-tuple jobs, one for the joins on relations with tuple width 1, and two or three jobs as required for the additional joins. Run the jobs in order three times, to collect three measurements for each join. Analyze the results as suggested in the previous sections.

7.16. XVI - Equijoin on Primary-Key Attribute

The form of the query is

```
SELECT ( target list ) FROM
     relation_name1, relation_name2 WHERE
     relation_name1.KEY=relation_name2.KEY
```

The purpose of the benchmark is to determine whether a simple merge strategy is used when two relations are joined on primary key. This is the most efficient strategy for this type of join, since the tuples are ordered by primary key. Again, we require that the relations being joined have the same tuple width. The target list and the join predicate should be constructed following the guidelines given in Benchmark XV.

The set of joins for this benchmark is the same as the set of joins for Benchmark XV. The jobs from Benchmark XV can be modified for this benchmark. Run the jobs three times. Calculate the mean elapsed time per operation using the method determined in Benchmark XV. Plot the results from this benchmark on copies of the graphs from Benchmark XV. If the mean elapsed times for Benchmark XVI are lower than the mean elapsed times for Benchmark XV, the next step is to determine whether a merge strategy is being used.

The time complexity of a merge is O(2*(C(S)+C(T))-1). In this particular case, we can terminate the merge as soon as all of the tuples of the smaller relation have been matched, so the complexity is O(2*min{C(S),C(T)}-1). So the number of operations required is 2*min{C(S),C(T)}-1. However, we expect that if a merge strategy is being used, the queries will be basically access-intensive. Therefore, compute the mean elapsed time per relevant block for each join. The number of relevant blocks for a join is the sum of the numbers of relevant blocks in each relationW

Graph the mean elapsed time per relevant block, using number of relevant blocks.as the unit on the x-axis. If the queries are, as expected, access-intensive, the curve will be increasing.

7.17. XVII - Equijoin on Primary-Key Attribute with Alternative Placement Strategy

The form of the query is the same as that for Benchmark XVI. This is a limited benchmark designed for those machines where the user can specify the placement of the relations across the devices in the database store. Recall that, initially, the database was assumed to reside on a single volume. For this benchmark, the relation M1 must be moved to a different volume.

This benchmark will include the joins S1 X.M1, and M1 X L1. A job from Benchmark XVI can be modified for this benchmark. Run the job three times. Compute the mean elapsed times per operation using the method determined in Benchmark XV. Graph the mean elapsed times for the two joins from Benchmark XVI. Then graph the mean elapsed times for the two joins from the current benchmark.

Compare the results. If a significant improvement results from the alternative placement, move relations M2, M3, and M4 to a different volume. In this case the measurements from this benchmark will become the <u>baseline</u> <u>measurements</u> for joins S1 X M1 and

M1 X L1. Otherwise, the measurements from Benchmark XVI will serve as the baseline measurements. Guidelines for query classification are the same as those for Benchmark XV.

Since the joins for this benchmark are the same as the joins from Benchmark XVI, no additional queries will be added to the query pools. However, reclassification of the join queries for S1 X M1, and M1 X L1 may be a result of this benchmark.

7.18. XVIII - Equijoin on Primary-Key Attribute With Join Predicate Reversed

The form of the query is

```
SELECT (target list) FROM
     relation_name1, relation_name2 WHERE
     relation_name1.KEY=relation_name2.KEY.
```

The purpose of the benchmark is to discover whether the machine recognizes that it is most efficient to select the relation with the smaller cardinality as the source relation. When the volume of the relations being joined exceeds the primary memory capacity, it is more efficient to select the smaller relation as the source relation. This minimizes the I/O overhead. When the machine does not have primary memory, the tuples of one of the relations must be sent to all of the processors. Sending the tuples of the smaller relation will minimize communication overhead.

The target list should include the names of all of the attributes of both relations. The join predicate should be constructed such that relation_name1 is the relation with the larger cardinality.

This benchmark will include two joins. The two joins are those identified in Benchmark XV such that the volume of the larger relation is equal to or greater than the primary memory capacity or the amount of data which can be accessed simultaneously from the database store. One multiple-tuple job is required to execute the joins. Run the job three times.

Graph the mean elapsed times per operation for these joins from Benchmark XVI. Then graph the mean elapsed times per operation from the current benchmark. If the machine recognizes that the smaller relation should be selected as the source relation, the means will be very close. A significant increase in mean elapsed times may indicate that the first named relation in the join predicate is used as the source relation. This benchmark is included for comparative purposes only. The queries will not be assigned to any query pool.

7.19. XIX - Equijoin on Primary-Key Attribute with Selection on Secondary-Key Attribute

The form of the query is

```
SELECT (target list) FROM
    relation_name1, relation_name2 WHERE
    relation_name1.P5A="value" AND
    relation_name2.P5A="value" AND
    relation_name1.KEY=relation_name2.KEY
```

The purpose of this benchmark is to determine whether selection is performed before join. The values for the selection predicates, relation_name1.P5A="value" and relation_name2.P5A="value", should in each case be the value of P5A for the first 5% of the tuples in the relation. This insures that the key values for the relevant tuples in the two relations will be in the range [0,.05*C(x)] The benchmark will include three joins, S1 X M1, S1 X L1, and M1 X L1. These joins are chosen so that we may compare the measurements from this benchmark to the baseline measurements for these joins.

The selection predicates, will reduce the number of operations required for the joins. When a straightforward join algorithm is used, the number of operations required will be (0.05)*C(S) * (0.05)*C(T) = (0.0025)*C(S)*C(T). When a sort-and-match algorithm or a hashing algorithm is used, the number of operations required will be the number of operations required for joining all of the tuples in the source and target relations multiplied by (0.05). For this query and this database structure, it is always desirable to perform the selection before the join. This is not desirable in all cases. Appendix D contains an analysis of performing selection before join.

One multiple-tuple job is required to execute the joins. Run the job three times. Graph the baseline measurements for these joins, using the sum of the cardinalities of the source and target relations as the unit on the x-axis. Next, compute and graph the expected mean elapsed time per operation, computed as indicated above. Finally, compute and graph the mean elapsed times per operation from the current benchmark. The guidelines for query classification given for Benchmark XV apply.

If selection is performed before join, the means from this benchmark should be close to the expected means. If instead the means are closer to the baseline measurements, this may indicate that join is performed before selection.

7.20. XX - Semijoin on Primary-Key Attribute with Selection on Secondary-Key Attribute

The form of the query is

```
SELECT (target list) FROM
    relation_name1, relation_name2 WHERE
    relation_name1.P5A="value" AND
    relation_name2.P5A="value" AND
    relation_name1.KEY=relation_name2.KEY
```

The purpose of this benchmark is to compare the performance of the machine for semijoins with the performance of the machine for equijoins. The target list will contain the names of all of the attributes of relation_name1. The value for the selection predicate should be the value of P5A for the first 5% of the tuples in the relation. The join predicate should be constructed so that relation_name2 is the name of the relation with the smaller cardinality. The joins for this benchmark are S1 X M1, S1 X L1,and M1 X L1. This is the same set of joins used in Benchmark XIX.

Therefore we have the measurements from Benchmark XIX for comparison.

One multiple-tuple job is required. Run the job three times. Graph the mean elapsed times per operation from the previous benchmark, Benchmark XIX. Add to the graph the mean elapsed times per operation from the current benchmark, computing the number of operations in the same manner as for the previous benchmark. If the curves are very close, the machine may use the same algorithm for join and semijoin. If the curve for this benchmark is significantly lower than the curve for the previous benchmark, the machine may be using a different algorithm for semijoin.

Let us consider an algorithm for semijoin. Suppose that the relevant tuples from relation_name2 are selected and read from the secondary store. The KEY values from those tuples are then used to select the relevant tuples from the relation_name1. The tuples for the result relation are then selected from the relevant tuples from relation_name1. When this algorithm is used, the number of operations required is 2 * .05*C(relation_name2). It is important to consider that the operations in this case are accesses to the database store. The work required of the processor is limited to applying the selection predicates. We expect that the performance should reflect that the queries are access-intensive. Therefore, instead of mean elapsed times per operation, we will compute mean elapsed times per relevant block. The total number of relevant blocks for a join is twice the number of relevant blocks of relation_name2.

Graph the mean elapsed times per relevant block for this benchmark, using the number of relevant blocks as the unit on the x-axis. Where the curve is parallel to the x-axis, the mean elapsed times are a constant function of the number of relevant blocks. This indicates that the queries are access-intensive. Where the curve rises, the overhead of accessing and searching the index is affecting the mean elapsed times. In this case, the queries are overhead-intensive.

7.21. XXI - Semijoin on Primary-Key Attribute With Selection on Secondary-Key Attribute Using the ANY Set Operator

The form of the query is

```
SELECT * FROM relation_name1 WHERE
   relation_name1.P5A="value" AND
   relation_name1.KEY = ANY
      (SELECT KEY FROM relation_name2
       WHERE relation_name2.P5A="value")
```

This is a limited benchmark to determine whether it is more efficient to construct semijoins using the ANY set operator. If the analysis of Benchmark XXI indicates that the semijoins are executed using the same algorithm as equijoins, this benchmark should be executed. Otherwise this benchmark may be skipped. Relation_name2 should be the name of the relation with the smaller cardinality. The value for the selection predicates should be the value of P5A for the first 5% of the tuples in the relation.

Graph the mean elapsed time per relevant block in the same manner as that indicated for the semijoin algorithm in Benchmark XX. The same guidelines for interpretation apply.

7.22. XXII --Inequality Semijoin on Non-Key Attribute With Selection on Secondary-Key Attribute

The form of the query is

```
SELECT (target list)  FROM
  relation_name1, relation_name2  WHERE
      relation_name1.P5A="value" AND
      relation_name2.P5A="value" AND
      relation_name1.KEY != relation_name2.KEY
```

The purpose of this benchmark is to measure the performance of the machine for inequality joins. The target list should contain the names of all of the attributes of relation_name1. The name of the relation with the smaller cardinality should be substituted for relation_name1. The value for the selection predicates should be the value of P5A for the first 5% of the tuples in the relation. The joins for this benchmark are S1 X M1, S1 X L1, and M1 X L1. This set of joins was chosen to show the effect of increasing cardinality on the performance of inequality join.

Inequality joins can only be accomplished by comparing all of the relevant tuples in the target relation to all of the relevant tuples in the source relation. Therefore the number of operations required for the inequality join is O(C(S)*C(T)). If the performance is a constant function of the number of operations, then the mean elapsed time per operation will be a constant function of the number of operations. The queries above specify an inequality join over 5% of the tuples of the source relations and 5% of the tuples of the target relations. Therefore, the number of operations required for these joins will be (0.0025)*C(S)*C(T).

However, we must also consider that the size of the result set for these inequality joins will have greater cardinality than the sum of the number of tuples in the source and target relations. We can show this as follows. The size of the result relations for these inequality joins is O(C(S)*C(T) - min{C(S),C(T)}), the difference between the product of the cardinalities of the source and target relations and the cardinality of the result relation of an equality join. We must prove that

$$C(S)*C(T) - \min\{C(S),C(T)\} > C(S) + C(T).$$

The proof in Appendix E shows that this is the case whenever C(S) > 2. The implication that we can draw is that the performance of the machine for inequality joins may be proportional to the size of the result relation. Since all the relations in the joins for this benchmark have the same tuple width, the variation in the size of the result relation will be proportional to the cardinality of the result relation. For these inequality joins, with selection of only 5% of the tuples of the source and target relations, the cardinality of the result relation will be (0.0025)*C(S)*C(T) - (.05)*min{C(S),C(T)}.

One multiple-tuple job is required. Run the job three times. First compute and graph the mean elapsed times per operation. If the curve is parallel to the x-axis, the mean elapsed times are a constant function of the number of operations, and the queries are processor-intensive. If, however, the curve rises, the mean elapsed times may be a function of the size of the result relation.

To determine whether the mean elapsed times are a function of the cardinalities of the result relations, first compute the mean elapsed times per tuple of the result relation. Then graph the mean elapsed times, using the cardinality of the result relation as the unit on the x-axis. If the curve is parallel to the x-axis or increasing, the mean elapsed times are a constant function of the size of the result relation. In this case, we will classify the queries as overhead-intensive.

7.23. XXIII - 3-Way Semijoin on Primary-Key Attribute With Selection on Secondary-Key Attribute

There are two possible forms for the query. The first is

```
SELECT (target list) FROM relation_name1,
  relation_name2, relation_name3 WHERE
  relation_name1.P5A="value" AND
  relation_name2.P5A="value" AND
  relation_name2.P5A="value" AND
  relation_name1.KEY=relation_name2.KEY AND
  relation_name2.KEY=relation_name3.KEY.
```

The second is

```
SELECT * FROM relation_name1 WHERE
  relation_name1.P5A="value" AND
```

```
relation_name1.KEY =
  (SELECT KEY FROM relation_name2 WHERE
   relation_name2.P5A="value" AND
   relation_name2.KEY =
     (SELECT KEY FROM relation_name3 WHERE
      relation_name3.P5A="value) )
```

The purpose of the benchmark is to measure the performance of the machine for semijoins involving three relations. The form of the query depends on the results of previous benchmarks. If Benchmark XX indicates that a semijoin algorithm is used when the query is of the first form, the query for this benchmark should be of the first form. Otherwise the query should be of the second form. If the query is of the first form, the target list should be composed of the attribute names of all attribute values in a tuple of relation_name1. Also, the value chosen for the selection predicates should be the value of P5A for the first 5% of the tuples in the relation. In either form of the query, the name of the relation with the smallest cardinality should be substituted for relation_name3, the name of the relation with the next-smallest cardinality for relation_name2, and the name of the relation with the largest cardinality for relation_name1.

This benchmark includes three joins, S1 X S1 X M1, S1 X S1 X L1, and S1 X M1 X L1. Note that in each case the second and third relations in the join specification correspond to the semijoins executed in benchmarks XX and XXI. In order to avoid incurring overhead for joining a relation with itself, we will create a copy of relation S1 for the duration of this benchmark. We will refer to the copy of relation S1 as S1COPY, and rewrite the join specifications to be S1COPY X S1 X M1, etc.

One multiple-tuple job is required. Run the job three times. Graph and analyze the results using the guidelines from Benchmark XX.

7.24. XXIV - Single-Tuple Updates Qualified on Primary-Key Attribute

The form of the query is

```
UPDATE relation_name
   SET  RAND=RAND + 1
 WHERE relation_name.KEY = value.
```

The purpose of the benchmark is to measure the performance of single-tuple updates which cause no index modification as tuple width and cardinality vary.

This benchmark will be run against relations L1, L2, L3, L4, S4, and M4. Six jobs, one per relation will be required. Each job will execute sixty-two queries. The values for the qualification are selected randomly from a range of values from 0 to (C(relation_name)-1). when the machine has primary memory, a write-back policy may be used, where a block is written to the database store only when it is to be replaced in the primary memory. This makes interpretation of the measurements difficult.

If the machine caches data in primary memory, first run the job against relation L4. Discard the measurements from this job. Then, for all machines, run the six jobs, in the order S4, L4, M4, L2, L1, L3,. three times. The purpose of running the job against relation L4 before taking any measurements and running the jobs in the order specified is to equalize as much as possible the number of write-backs.

Graph the mean elapsed time per participating tuple from the four jobs against relations L1, L2, L3, and L4, using tuple width as the unit on the x-axis. Add to the graph the mean elapsed times from Benchmark V, Single-Tuple Selections Qualified on Primary Key Attribute. Graph the mean elapsed time per participating tuple from the three jobs against relations S4, M4, and L4, using cardinality as the unit on the x-axis. Add to the graph the mean elapsed times from Benchmark V. The elapsed times are plotted against tuple width and cardinality, since the number of participating tuples will in each case be 62, and the number of relevant blocks will be approximately 62, Add a line to each graph which represents twice the estimated access time per relevant block.

These queries are by definition access-intensive. Where the mean elapsed times are more than twice the estimated mean access time per relevant block, the means reflect the overhead of accessing and searching the index or some inequality in the number of write-backs. Where the mean elapsed times are less than twice the estimated mean access time, the means reflect a high number of cache hits or some inequality in the number of write-backs. The difference between the means from Benchmark V and the means from this benchmark represents the time required for writing the updated blocks back to the database store.

7.25. XV - Single-Tuple Deletes and Inserts Qualified on Primary-Key Attribute

We treat the deletes and inserts as a single benchmark. The benchmark is designed such that the inserts replace the records deleted. First we describe the benchmark for deletes. Then we describe the benchmark for inserts.

7.25.1. Single-Tuple Deletes

The form of the query is

```
DELETE FROM relation_name WHERE
     relation_name.KEY=value.
```

The purpose of this benchmark is to measure the performance of the machine for deletions. It is important to note that, when a tuple is deleted, three indices must be modified: the primary index on KEY, the secondary index on *MIRROR*, and the secondary index on P5A.

This benchmark will be run against relations L1, L2, L3, L4, S4, and M4. Six jobs, one per relation, will be required. Each job will execute sixty-four queries. The values for the qualification are selected randomly from a range of values from 0 to (C(relation_name)-1). The measurements from the first four jobs will show the effect of increasing tuple width on mean elapsed times. The measurements from the last three jobs will show the effect of increasing cardinality on mean elapsed times.

Run the jobs in the same order as specified for Benchmark XXIV. Graph the mean elapsed time per participating tuple from the four jobs against relations L1, L2, L3, and L4, using tuple width as the unit on the x-axis. Graph the mean elapsed time per participating tuple from the three jobs against relations S4, M4, and L4, using cardinality as the unit on the x-axis. Add to each graph the measurements from Benchmark XXIV.

These queries are by definition overhead-intensive. The difference between the means from Benchmark XXIV and the means from the current benchmark represent the overhead of updating three indices. There may also be overhead from reorganizing the physical data.

7.25.2. Single-Tuple Inserts

The form of the query is

```
INSERT INTO relation_name
     VALUES (list of values).
```

The list of values should be constructed such that the tuples inserted replace exacty the tuples previously deleted. The purpose of this benchmark is to measure the performance of the machine for insertions.

The job structure and the sequence of execution are the same as those for the delete benchmark. Graph the mean elapsed times in a similar manner. Compare the mean from this benchmark with the means from the delete benchmark.

8. MULTIPLE-USER BENCHMARKS

Benchmarks which involve multiple jobs will give a more realistic picture of database machine performance. It is for this reason that we include in our methodology a method for constructing job mixes for multiple-user benchmarks. The performance indices of the multiple-user benchmarks will be compared to the performance indices derived from the single-user benchmarks as described in Chapter 4. The multiple-user benchmarks will be a collection of jobs. There are three standard jobs; one for access-intensive queries, one for processor-intensive queries, and one for overhead-intensive queries.

The method for constructing the standard jobs is described in Section 8.1. Concurrent operations will affect the performance of a particular job mix. In Section 8.2, we describe three properties for a job mix, conflict, contention, and sharing, and give a method for estimating the probability of each. The procedure for executing the job mixes is given in Section 8.3. In Section 8.4, we suggest some particular standard job mixes and discuss the construction of job mixes to fit a particular collection of applications.

8.1. Construction of the Standard Benchmark Jobs

The standard benchmark jobs are modeled as a collection of queries and query streams from one of the query pools from the basic benchmarks. A query is a multiple-tuple query. A query stream is a collection of single-tuple queries against a single relation. A particular job is characterized as access-intensive, overhead-intensive, or processor-intensive, depending on the query pool from which the queries and query streams are selected.

First we select from each query pool a small set of queries and query streams such that the mean of the mean elapsed times per participating tuple is very close to the mean elapsed time index for that query pool. We need a method for selecting this set of queries and query streams. A brute-force method of selecting the queries would involve a polynomial time algorithm, with time complexity O(n**n) for selecting sets of n queries. Clearly this is not practical. We suggest instead that the means be analyzed for any clustering tendencies. We expect the means from similar queries which exhibit similar behavior to be very close. Queries whose means are very close form a <u>cluster</u>.

Where the number of means is small, it is possible to identify clusters by examination. Arrange the means per participating tuple for a query pool in ascending order, and partition them into clusters based on closeness. This method is illustrated in Chapter 9. Where the number of means is large, a cluster analysis tool may be used. A brief discussion of formal cluster analysis algorithms and tools is included in Section 8.1.1.

In the discussion which follows, a <u>query mean</u> is defined to be the mean elapsed time per participating tuple for a query or query stream. A <u>cluster mean</u> is defined to be the mean of the query means in the cluster, where a <u>cluster</u> is a group of queries whose elapsed times per participating tuple are very close. A <u>query elapsed time mean</u> is the mean elapsed time per query over the repetitions of the same query.

For each query pool, do the following. Assume that cluster analysis is complete. The result is that the query means, and by implication the queries and query streams, have been grouped into clusters. For each cluster, the cluster mean has been computed. Now, find the smallest group of clusters such that the mean of the cluster means is very close to the elapsed time index for the query pool. This is a group and not a set in the mathematical sense, since a particular cluster may occur more than once in the group.

Then select from these clusters the smallest group of queries and query streams such that the mean of the query means is very close to the elapsed time index for the query pool. Again, the same query or query stream may occur more than once in the group. This group of queries and query streams will be the basis for constructing the standard job.

Now we will determine the approximate length of time each query in each job should run, so that the jobs will be of approximately equal length. Let a, p, and o be the numbers of queries and query streams in the group selected for the access-intensive job, the processor-intensive job, and the overhead-intensive job respectively. Let N = max{a,p,o}. Let M be the mean of the N query elapsed time means. Compute the total job elapsed time, J = N*M. Each query and query stream in the access-intensive job should run for approximately J/a minutes. Each query and query stream in the processor-intensive job should run for approximately J/p minutes. Each query and query stream in the overhead-intensive job should run for approximately J/o minutes. This method is adopted in an attempt to preserve the essential performance characteristics of the various queries while restricting the job run time to some reasonable limit. If total job elapsed time is unreasonably high, consider using some figure less than the calculated M.

We will give an example of calculating total job elapsed time. A reasonable heuristic for reducing total job elapsed time is used. The total job elapsed time will be computed for the lists of queries in Figure 23. Each query is represented by the set of relations which the query accesses. Within each query pool, the queries are assigned identifiers. Queries with identifiers of the form Mi are queries from multiple-tuple jobs. Queries with identifiers of the form Sj are query streams from single-tuple jobs. In this case, the group of access-intensive

Access-Intensive Queries:

M1={S1},M2={S1},M3={S2},M4={S2},M5={S1,M1},
M6={S1,M1},M7={S3},M8={M1}

Processor-Intensive Queries:

M1={S1},M2={S1,M1}

Overhead-Intensive Queries:

S1={S1},S2={S1},S3={S2},M1={L1},M2={L2}

Figure 23: Example Query Sets for the Standard Job Types

queries includes eight queries. There are two processor-intensive queries and five overhead-intensive queries. Then N = max{8,2,5} = 8. Let M, the mean of the query elasped times for the 8 access-intensive queries, be 18.75 mintues. Then the total job elapsed time is N*M = 150 minutes. If total job elapsed time is 150 minutes, we know that the job run time will be more than three hours. This is somewhat long. Let us find a way to reduce the total job elapsed time.

Suppose that the minimum query elapsed time mean for the access-intensive queries is 15 minutes, and that the maximum query elapsed time mean is 45 minutes. If we let M = 15 minutes, then the total job elapsed time is N*M = 120 minutes. Examine the query with the maximum query elapsed time mean of 45 minutes. If this query returns a large number of tuples, the basic performance characteristics for this query will probably not be altered if the query elapsed time is limited to fifteen minutes. So let us use 120 minutes as the total job elapsed time. Then each query in the access-intensive job should run for 120/8=15 minutes. Each query in the processor-intensive job should run 120/2=60 minutes. Each query in the overhead-intensive job should run 120/5=24 minutes. Figure 24 is an abstract representation of the sequences of queries executed in each standard job type over time.

The job model for the multiple-user jobs will be a composite of the single-tuple and multiple-tuple job models presented in Chapter 5. Each job must have the capability to execute both single-tuple query streams and multiple-tuple queries. The queries and query streams will be selected randomly without replacement from the group of queries and query streams which comprise the job. Some capability to cancel execution of a query is assumed, since such a capability will be required when limiting execution of a query to a specified length of time.

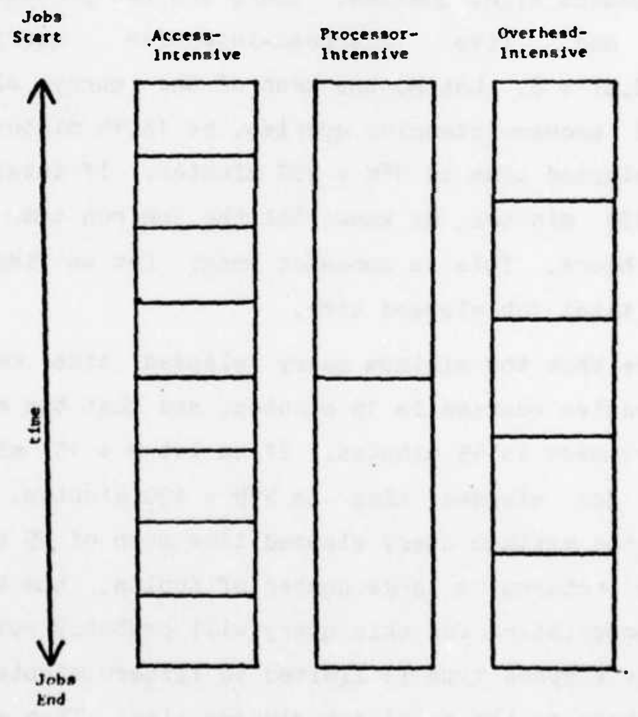


Figure 24: Sequences of Queries of the Standard Job Types Over Time

Limiting the execution time for a query stream can be accomplished by checking the elapsed time after every query. Limiting the execution time for a multiple-tuple query is more complex. We suggest that some method like the following be used. Check the clock after every 100 tuples returned from the oatabase machine, and after the final tuple has been returned. If the elapsed time is greater than the limit, terminate execution. If the elapsed time is less than the limit, and the final tuple has not been returned, continue retrieving tuples. If the elapsed time is less than the limit, and the final tuple has just been returned, then determine whether the the remaining time is less than 10% of the elapsed time limit. If so, terminate execution; otherwise execute the query again. The correspondence shown between jobs over time will vary from that shown in Figure 24 due to the overhead required for limiting the elapsed time per query.

8.1.1. Cluster Analysis

A cluster analysis "groups objects or data items, according to indices of "alikeness" between pairs of objects" [Dube80]. One type of clustering is called <u>partitional clustering</u>. A partitional clustering analysis can be used to divide a set of n patterns or sets of measurements, $\{x_1, x_2, \ldots, x_n\}$, into exactly C disjoint subsets, where C<<n. Two well-known partitional clustering programs are FORGY [Forg65] and ISODATA [Ball65]. These are partitional clustering methods which use the Euclidean distance as the distance metric. In our case, where a single variable is measured, the Euclidean distance is defined $d(i,j)=sqrt[|x_i-x_j|**2]=|x_i-x_j|$, where $i < j <= n$. These algorithms require an initial clustering. For example, C cluster centers or seed points must be specified by the user or chosen at random. The desired number of clusters is usually specified a priori. The algorithms contain heuristics for eliminating "outliers", data points which do not conform with the rest of the patterns. Some implementations of FORGY generate a fixed number of clusters; other implementations allow merging and splitting of clusters, so that the number of resulting clusters may differ from the initial C. ISODATA tries to establish a user-specified number of clusters, unless it finds this number to be unrealistic. The references given above may be consulted for more details.

Another clustering method is hierarchical clustering. This method creates a nested sequence of partitions from a proximity matrix. The proximity matrix is a symmetric matrix which shows the proximity of pairs of patterns or measurements. The Euclidian distance can be used as a metric for constructing the proximity matrix. Hierarchical clustering methods can be used to map proximity matrices into dendograms. Figure 25 shows a dendogram constructed using a hierarchical clustering method. Figure 26 shows a proximity dendogram constructed using the same method. The levels of the clustering for this dendogram are based on the proximity value at which clustering first occurs. The number of levels between the merging of two queries is an indication of the closeness of the cluster means. A number of standard statistical

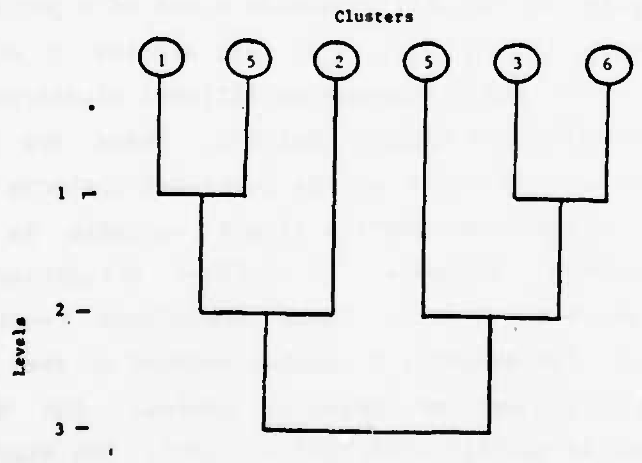


Figure 25: A Dendogram

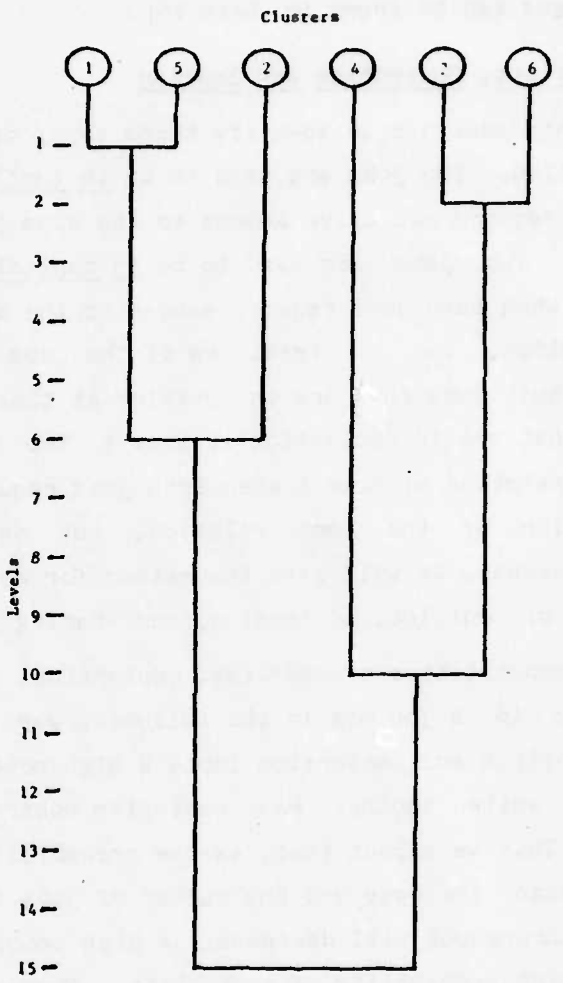


Figure 26: A Proximity Dendogram

packages contain hierarchical clustering programs. Among these are BCTRY, BMDP CLUSTAN, NT-SYS, and OSIRIS. A critique of various packages can be found in [Blas75].

8.2. Conflict, Contention and Sharing

In this analysis we identify three types of concurrent access to a relation. Two jobs are said to be in conflict at time t when both jobs require exclusive access to the same portion of the same relation. Two jobs are said to be in contention for a relation at time t when both jobs require access to the same portion of the same relation, and at least one of the jobs requires exclusive access. Thus, jobs that are in conflict at time t are a subset of the jobs that are in contention at time t. Two jobs are said to be sharing a relation at time t when both jobs require access to the same portion of the same relation, but neither job requires exclusive access. We will give the method for estimating the probabilities of conflict, contention, and sharing in Section 8.2.1.

The probabilities of conflict, contention, and sharing affect performance in a job mix in the following ways. High probabilities of conflict and contention imply a high probability that one job waits while another has exclusive control of a block of a relation. Thus we expect that, as the probabilities of conflict and contention increase and the number of jobs in the mix is held constant, throughput will decrease. A high probability of sharing implies a high probability of cache hits. Thus we expect that, as the probability of sharing increases and the number of jobs in the mix is held constant, throughput for machines which cache data will increase.

It is important to note that the probability estimates reflect the possibility that any two jobs are in conflict, in contention, or sharing at time t. In reality, more than two jobs may be in conflict (or in contention or sharing) at time t. Upper and lower bounds can be established for the probabilities of conflict, contention, and sharing among three or more jobs. However, the computation of the conditional probabilities involved is complex. We feel that the work involved is disproportionate to the value of the index.

8.2.1. Estimating the Probabilities of Conflict, Contention, and Sharing

A relation consists of a set of p blocks, $\{b_1, b_2, \ldots, b_p\}$. Let the probability of selecting the kth block, $1 <= k <= p$, be $(1/p)$. Given a set of m relations, $\{r_1, r_2, \ldots, r_m\}$, let the probability of selecting the jth relation, $1 <= j <= m$, be $(1/m)$. If the jth relation has p blocks, then the probability of selecting the kth block of the jth relation is $(1/m)*(1/p)$.

Now, given a list of n queries, $(q_1, q_2, \ldots, q_n)$, let the probability of selecting the ith query, $1 <= i <= n$, be $(1/n)$. If the ith query is over a set of m relations, the probability of selecting the jth of the m relations for the ith query is $(1/n)*(1/m)$. If the jth relation has p blocks, the probability of selecting the kth block in the jth relation for the ith query is $(1/n)*(1/m)*(1/p)$.

So, we can compute the probability of selecting a particular block in a particular relation for a particular query. Given two lists of queries, we can construct an algorithm which computes the probability that two queries, one from each list, will select the same block of the same relation. Figure 27 shows this algorithm, Concurrent_Access.

The lines of pseudo-code in Figure 27 are numbered on the left. Given a pair of lists of sets of relation names, the algorithm compares every set of relations names from the first list with every set of relation names from the second list. This is accomplished by the two for loops at lines 7 and 9. If any two queries access the same relation, the intersection of the sets of relation names will be non-empty. This case is tested at lines 11

```
1.  algorithm Concurrent_Access;

2.  variables
      P        : probability of concurrent access;
      R1,R2,S  : sets of relation names;
      i,j,k    : integers;
      R        : a relation name;
      L1,L2    : lists of sets of relation names;

3.  functions
      $l(Lx)$    : returns number of sets of
                   relation names in the list Lx;
      $s(Lx,i)$  : returns the ith set of
                   relation names in the list Lx;
      $r(S,k)$   : returns the kth relation name
                   in the set S;
      $c(Rx)$    : returns the cardinality of the
                   set of relation names Rx;
      $b(R)$     : returns the number of blocks
                   in relation R;

4.  begin Concurrent_Access;

5.      P <- 0;
6.      input(L1,L2);

7.      for i <- 1 to $l(L1)$ do;
8.        R1 <- $s(L1,i)$;

9.        for j <- 1 to $l(L2)$ do;
10.         R2 <- $s(L2,j)$;
11.         S <- $R1 \cap R2$;
12.         if $S \neq \emptyset$ then
13.           for k <- 1 to $c(S)$ do;
14.             R <- $r(S,k)$;

15.             P <- $P + (c(R1))^{-1} *$
                        $(c(R2))^{-1} *$
                        $(b(R))^{-1}$;
16.           end for;
17.         end if;
18.       end for;
19.     end for;

20.     P <- $P * (l(L1))^{-1} *$
                 $(l(L2))^{-1}$;
21.     output((L1,L2),P);

22. end Concurrent_Access;
```

Figure 27: Concurrent_Access Algorithm

and 12. If the intersection of the sets of relations is not empty, then the probability that the two queries are accessing the same block of the same relation at time t is calculated for each relation in the intersection of the sets. A sum of these probabilities is accumulated. This is shown in the loop at lines 13 through 16. After all the pairs of sets of relations have been compared, the probability that two queries access the same block of the same relation at time t is calculated. This is the sum of the probabilities accumulated within the loops multiplied by the probabilities of selecting a particular query from the first list and a particular query from the second list. This is shown in Figure 27 at line 20.

Let A be the list of sets of relations for the standard access-intensive job queries. Let P and O be the lists of sets of relations for the standard processor-intensive and overhead-intensive job queries, respectively. If we input any two of these lists to the Concurrent_Access algorithm, the algorithm will compute the probability that two queries, one from each list, will select the same block of the same relation.

However, in order to estimate the probabilities of conflict, contention, and sharing, we must consider those queries which require exclusive access to data separately from those which do not require exclusive access. First, let us partition each of the lists A, P, and O into two lists on the basis of whether the query requires exclusive access to data. Let us call the six lists which result from this partitioning A_e, A_s, P_e, P_s, O_e, and O_s. The subscript 'e' indicates that exclusive access is required. The subscript 's' indicates that exclusive access is not required, i.e., data may be shared. We will call the lists A, P, and O, the parent lists of A_e and A_s, P_e and P_s, and O_e and O_s, respectively.

Now let us revise the Concurrent_Access algorithm in the following manner. Define a new function $\ell'(Lx)$ which, given the name of a list as an argument, returns the number of queries in the parent list of Lx. Substitute $\ell'(Lx)$ for $\ell(Lx)$ in the Concurrent_Access algorithm. The revised algorithm is shown in Figure 28. We call this the Revised_Concurrent_Access algorithm. Given a pair of lists from the six lists defined above, the Revised_Concurrent_Access algorithm computes the probability that two queries, one from each list, will access the same block of the same relation.

We can compute estimates of the probabilities of conflict, contention, and sharing for a job mix in the following manner. Let the job mix be a mix of A access-intensive jobs, P processor-intensive jobs, and O overhead-intensive jobs. We can characterize the job mix as a list of jobs, for example

```
MIX = (AJOB1,AJOB2, ...AJOBA,
       PJOB1,PJOB2, ...PJOBP,
       OJOB1,OJOB2, ...OJOBO).
```

Let us introduce two sets which will be used in expressing the formulae for the probabilities.

$$S_e = \{ (A_e,A), (P_e,P), (O_e,O) \}$$
$$S_s = \{ (A_s,A), (P_s,P), (O_s,O) \}$$

Each 2-tuple specifies the name of a list of sets of relations and the number of related jobs. For example, the 2-tuple (A_e,A) is the name of the list of sets of relations for access-intensive queries which require exclusive access to data. A is the number of access-intensive jobs in the mix.

We can calculate the probability of conflict for the mix as follows. The probability that one pair of access-intensive jobs is in conflict is $P(A_e,A_e)$, where $P(A_e,A_e)$ is the output of the Revised_Concurrent_Access algorithm for the pair of lists (A_e,A_e). If there are $A > 1$ access-intensive jobs in the mix, then there are $\mathcal{C}(A,2) = (A!)/(2!*(A-2)!)$ possible pairs of access-intensive jobs, where $\mathcal{C}(n,r)$ is the number of combinations of n things taken

```
1.  algorithm Revised_Concurrent_Access;

2.  variables
      P        : probability of concurrent access;
      R1,R2,S  : sets of relation names;
      i,j,k    : integers;
      R        : a relation name;
      L1,L2    : lists of sets of relation names;

3.  functions
      $l'(Lx)$   : returns number of sets of
                   relation names in the parent
                   list of the list Lx;
      $s(Lx,i)$  : returns the ith set of
                   relation names in the list Lx;
      $r(S,k)$   : returns the kth relation name
                   in the set S;
      $c(Rx)$    : returns the cardinality of the
                   set of relation names Rx;
      $b(R)$     : returns the number of blocks
                   in relation R;

4.  begin Revised_Concurrent_Access;

5.      P <- 0;
6.      input(L1,L2);

7.      for i <- 1 to $l'(L1)$ do;
8.        R1 <- $s(L1,i)$;

9.        for j <- 1 to $l'(L2)$ do;
10.         R2 <- $s(L2,j)$;
11.         S <- $R1 \cap R2$;
12.         if $S \neq \emptyset$ then
13.           for k <- 1 to $c(S)$ do;
14.             R <- $r(S,k)$;

15.             P <- P + $(c(R1))^{-1}$ *
                         $(c(R2))^{-1}$ *
                         $(b(R))^{-1}$;
16.           end for;
17.         end if;
18.       end for;
19.     end for;

20.     P <- P * $(l'(L1))^{-1}$ *
                 $(l'(L2))^{-1}$;
21.     output((L1,L2),P);

22. end Revised_Concurrent_Access;
```

Figure 28: Revised_Concurrent_Access Algorithm

r at a time. Then the probability of conflict between all possible pairs of access-intensive jobs in the mix is

$$C(A,2) * P(A_e,A_e)$$

The probability of conflict between one pair of processor-intensive jobs is $P(P_e,P_e)$, the output of the Revised_Concurrent_Access algorithm for the pair of lists (P_e,P_e). With $P > 1$ processor-intensive jobs in the mix, the probability of conflict between all possible pairs of processor-intensive jobs in the mix is

$$C(P,2) * P(P_e,P_e)$$

The probability that one access-intensive job is in conflict with one processor-intensive job is $P(A_e,P_e)$, the output of the Revised_Concurrent_Access algorithm for the pair of lists (P_e,P_e). There are $(A * P)$ possible processor-intensive-access-intensive pairs of job in the mix. The probability of conflict between all pairs of access-intensive and processor-intensive jobs in the mix is

$$A * P * P(A_e,P_e)$$

We can compute in a similar manner the probability of conflict between all pairs of overhead-intensive jobs, all pairs of access-intensive and overhead-intensive jobs, and all pairs of processor-intensive and overhead-intensive jobs. The estimated probability of conflict for the mix is the sum of all of these probabilities.

Figure 29 shows an algorithm for calculating the estimated probability of conflict for a job mix, using the Revised_Concurrent_Access algorithm as a function. This algorithm is called Single_Set, since it deals with only one of the sets S_e or S_s. In this case we are dealing with the set S_e. Given the set S_s as input, the loop at lines 7 through 10 calculates the

```
1.  algorithm Single_Set;
2.  variables
        SS       : set of 2-tuples of the
                   form (S,x) where S is the
                   name of a sublist and x is
                   the number of related jobs;
        (S1,x)   : 2-tuples from the sublist;
        (S2,y)   : 2-tuples from the sublist;
        E        : estimator of probability;
        i        : integer;
3.  functions
        t(SS,j)  : returns the jth 2-tuple from
                   the sublist SS;
        F(S1,S2) : returns the result of the
                   Revised_Concurrent_Access
                   algorithm applied to sublists
                   S1 and S2;

4.  begin Single_Set

5.      E <- 0;
6.      input(SS);

7.      for i <- 1 to 3 do;
8.          (S1,x) <- t(SS,i);
9.          E <- E + $\mathcal{C}_{x,2}$ * F(S1,S1);
10.     end for;

11.     for i <- 1 to 2 do;
12.         (S1,x) <- t(SS,i);
13.         for j <- i+1 to 3 do;
14.             (S2,y) <- t(SS,j);
15.             E <- E + x*y*F(S1,S2);
16.         end for;
17.     end for;

18.     output(E);

19. end Single_Set;
```

Figure 29: Single_Set Algorithm

relevant blocks of the relations will usually vary. More accesses will be required for the relations with the larger number of relevant blocks. Third, we have made an assumption of the uniform probability of selecting any block from a relation. Since the relevant blocks may be limited by selection, some blocks of the relation would in reality never be accessed.

A better estimator would take these factors into account. However, since in all cases the job mixes for a set of multiple-user benchmarks will be based on the same standard benchmark jobs, the comparability of the estimators across different job mixes on the same machine is maintained. The comparability of the estimators across sets of benchmarks for different machines and different machine configurations will not be maintained. However, since the queries for the standard benchmark jobs in each case are selected from the same set of standard single-user benchmark queries, and the estimators are computed in a uniform manner, some relatedness may be assumed.

8.3. A Procedure for Running Multiple-User Benchmarks

A mix of standard benchmark jobs should be executed with no workload on the host system other than that required to support the database machine users. The jobs should be allowed to run to completion. We recommend that each mix be run twice, to verify reproducibility of results. After the benchmarks have been completed, performance indices should be calculated as specified in Chapter 4.

To determine the effect of interactive users, the same mix should be executed with a pre-determined number of interactive users. The scripts for the interactive users should be constructed as random selections of any query from benchmark V, IX, X, and XXIV, any query against relations S1 and S2 from benchmarks II, III, IV, XII, XIII, and XIV, and the semijoin query for S1 X M1 from benchmark XX or XXI. The queries from benchmarks V,

```
1.  algorithm Double_Set;

2.  variables

        SS1,SS2 : sets of 2-tuples of the
                  form (S,x) where S is the
                  name of a sublist and x is
                  the number of related jobs;
        (S1,x)  : 2-tuples from the sublists;
        (S2,y)  : 2-tuples from the sublists;
        E       : estimator of probability;
        i       : integer;

3.  functions

        t(SS,j)  : returns the jth 2-tuple from
                   the sublist SS;
        P(S1,S2) : returns the result of the
                   Revised_Concurrent_Access
                   algorithm applied to sublists
                   S1 and S2;

4.  begin Double_Set

5.    E <- 0;
6.    input(SS1,SS2);

7.    for i <- 1 to 3 do:
8.        (S1,x) <- t(SS1,i);
9.        (S2,y) <- t(SS2,i);
10.       E <- E + $\mathcal{C}_{x,2}$ * P(S1,S2);
11.   end for;

12.   for i <- 1 to 3 do;
13.       (S1,x) <- t(SS1,i);
14.       for (j <- 1 to 3 and  j <>i)  do;
15.           (S2,y) <- t(SS2,j);
16.           E <- E + x*y*P(S1,S2);
17.       end for;
18.   end for;

19.   output(E);

20. end Double_Set;
```

Figure 30: Double_Set Algorithm

IX, X, and XXIV are either single-tuple queries or aggregations. The queries from benchmark II, III, IV, XII, XIII, and XIV are queries for 5% of the tuples. By limiting the queries selected from these benchmarks to the smaller relations, we limit the amount of data involved. The use of a terminal emulator is recommended, since the benchmark mixes are likely to run for several hours.

8.4. Selecting Job Mixes

The performance of the machine for a job mix depends not only on the composition of the mix, but also on the number of jobs in the mix. Say that we select a basic mix. By doubling the number of jobs in the mix and maintaining the proportionality of access-intensive, processor-intensive, and overhead-intensive jobs, we can isolate the effect of adding users to the machine. By varying the composition of the mix while holding the number of jobs constant, we can determine the effect of varying compositions. Doing both may require an unacceptably large number of benchmarks.

Instead, we suggest a set of four standard benchmarks, with the composition shown in Table 5. Four standard job mixes are defined as sets of access-intensive (AJOBx), processor-intensive (PJOBx), and overhead-intensive (OJOBx) jobs. Thus the differences between the performance indices of MIX1 and MIX2 show the effect of adding one access-intensive job. The differences

```
MIX1 = {AJOB1, PJOB1, OJOB1}
MIX2 = {AJOB1, AJOB2, PJOB1, OJOB1}
MIX3 = {AJOB1, AJOB2, PJOB1, PJOB2, OJOB1}
MIX4 = {AJOB1, AJOB2, PJOB1, PJOB2, OJOB1, OJOB2}
```

Figure 31: Standard Job Mixes

between the performance indices of MIX2 and MIX3 show the effect of an additional processor-intensive job. The differences between the performance indices of MIX3 and MIX4 show the effect of an additional overhead-intensive job. The differences between the performance indices of MIX1 and MIX4 show only the effect of doubling the number of users.

These standard mixes should be executed first without interactive users, as specified in Section 8.3. Then execute the same mixes, each with the same number of interactive users. We recommend 6 interactive users, as a standard for machine-to-machine comparisons.

9. EXPERIMENTS IN BENCHMARKING A RELATIONAL DATABASE MACHINE

In this chapter, we report the results of some single-user benchmark experiments. The experiments were run during the course of the development of the methodology. They are similar to, if not identical to, those benchmarks specified in Chapter 7.

The measurements which we report here should not be taken as the ultimate performance of this particular machine, since later versions of the machine will undoubtedly perform better. The objective of performing the experiments was to learn how to construct effective benchmarks and to discover whether a meaningful interpretation could be made from the measurements. Thus we were not interested in the machine itself. Instead, we were interested in developing benchmarks and in interpreting measurements. We report these benchmarks to illustrate the methodology and to give some examples of interpretation. All of the measurements are shown in seconds or fractions of seconds.

In Section 9.1, we describe the database model used in the experiments. In Section 9.2, we describe the job model used in the experiments. We explain the difference between this job model and the batch job model proposed for this methodology. In Sections 9.3 through 9.8, we analyze experimental results for benchmarks I, II, XII, XIII, XIV, XV, XVI, XVIII, and XIX. The calculation of the performance indices is illustrated in Section 9.10. The method of constructing standard jobs for multiple-user benchmark job mixes is illustrated in Section 9.11.

9.1. The Experimental Database Model

The standard tuple templates for the four tuple widths are shown in Figure 32. The four tuple widths were chosen based on the

100-byte tuple width		200-byte tuple width		1000-byte tuple width		2000-byte tuple width	
ATTRIBUTE NAME	TYPE	ATTRIBUTE NAME	TYPE	ATTRIBUTE NAME	TYPE	ATTRIBUTE NAME	TYPE
key	i4	key	i4	key	i4	key	i4
mirror	c11	mirror	c11	mirror	c11	mirror	c11
rand	i4	rand	i4	rand	i4	rand	i4
uniqrand	i4	uniqrand	i4	chars	c63	chars	c79
chars	c4	chars	c4	p5	c9	p5	c9
letter	c1	letter	c1	p10	c9	p10	c9
p5	c9	p5	c9	p20	c9	p20	c9
p10	c9	p10	c9	p25	c9	p25	c9
p20	c9	p20	c9	p30	c9	p30	c9
p25	c9	p25	c9	p35	c9	p40	c9
p35	c9	p30	c9	p40	c9	p50	c9
p50	c9	p35	c9	p45	c9	p60	c9
p75	c9	p40	c9	p50	c9	p70	c9
p80	c9	p45	c9	p60	c9	p75	c9
		p50	c9	p65	c9	p80	c9
		p55	c9	p70	c9	p90	c9
		p60	c9	p75	c9	p100	c9
		p65	c9	p80	c9	up10	uc255
		p70	c9	p85	c9	up20	uc255
		p75	c9	p90	c9	up25	uc255
		p80	c9	p100	c9	up50	uc255
		p85	c9	up10	uc255	up75	uc255
		p90	c9	up25	uc255	up80	uc255
		p100	c9	up50	uc255	up100	uc255

Figure 32: The Standard Tuple Templates

2048-byte block size. We have a range of 20 tuples per block to 1 tuple per block. Each relation contains KEY, MIRROR, and RAND attribute values, and values for P5, which is equivalent to the P5A attribute value described in Chapter 2. There are no P5B, P5C, etc. attribute values. There is no COPY_KEY. When a benchmark requires an unindexed copy of the primary key attribute, we simulate the effect by destroying the primary index.

For our experiments, we used four cardinalities: 500 tuples, 1000 tuples, 2500 tuples, and 5000 tuples. Thus we have sixteen relations in our database. Table 2 shows the matrix of relations. The name of each relation, the volume of the relation (tuple width * cardinality) in millions of bytes, and the number of blocks in the relation are shown. Not all of the benchmarks reported here have been run on all of the relations or combinations of relations.

9.2. The Experimental Job Model

At the time we began the design of the experiments, in October, 1982, only an interactive user interface was available. We used an operating system service to divert output from the terminal to a spool file. A second operating system service was used to run the interactive interface from pre-constructed files, thus eliminating think time and entry time at the terminal.

The queries were in effect pre-compiled. This particular machine offered the facility to pre-parse queries and to store

		TUPLE WIDTH			
		1=100 bytes	2=200 bytes	3=1000 bytes	4=2000 bytes
CARDINALITY	S = 500 tuples	S1 .05 M 25 b	S2 .1 M 50 b	S3 .5 M 250 b	S4 1 M 500 b
	M = 1000 tuples	M1 .1 M 50 b	M2 .2 M 100 b	M3 1 M 500 b	M4 2 M 1000 b
	L = 2500 tuples	L1 .25 M 125 b	L2 .5 M 250 b	L3 2.5 M 1250 b	L4 5 M 2500 b
	G = 5000 tuples	G1 .5 M 250 b	G2 1 M 500 b	G3 5 M 2500 b	G4 10 M 5000 b

Table 2: Matrix of Relations in the Experimental Database

them in the database machine as "stored commands". The parsing of the query is then negligible. The transmission of the query to the database machine then involves a single execute token, along with the identifier for the stored command. These "stored commands" were created and destroyed as needed, so that at any one time, only a small number of commands were stored in the database.

Measurements were taken from the database machine clock before and after each query. The resolution of the clock is 1/60th second. Sampling the clock took, on the average over thirty successive experiments, 0.011 seconds. Given the elapsed time per query of the queries in this benchmark, this time is insignificant.

Our jobs were run through the interactive interface. Although the jobs had higher priority as a consequence, the measurements reflect the overhead of a terminal monitor. For this reason, we expect that measurements taken in a batch environment would be lower. The measurements reported here are the worst-case measurements for this machine, both because they were taken from an early version of the machine and because and interactive job model was used.

9.3. Benchmark I

This benchmark is titled "Unindexed Selection of 5% of Tuples". The measurements for all sixteen relations in the database are shown in Table 3. The mean elapsed times per relevant block are graphed against the number of relevant blocks in Figure 33. As expected, the mean elapsed times per relevant block are roughly a function of the number of relevant blocks. The means for the smaller numbers of relevant blocks reflect some overhead in setting up the query. However, we can consider all of the sixteen queries in this benchmark to be access-intensive. We will use the figures from this benchmark to estimate the time for accessing a block of data from the database store for comparison

Benchmark ID: I Benchmark Title: Unindexed Selection of 5% of Tuples

General Form of the Query: SELECT * FROM relation-name
WHERE P5B = "value"

Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ/RB	
I1	1	S1	100	500	3	2.4	.8			25	.032	
	2	M1		1000		4.5	1.5			50	.03	
	3	L1		2500		9	3			125	.024	
	4	G1		5000		15.3	5.1			250	.0204	
I2	5	S2	200	500		4.5	1.5			50	.03	
	6	M2		1000		9	3			100	.03	
	7	L2		2500		15.3	5.1			250	.0204	
	8	G2		5000		30	10			500	.02	
I3	9	S3	1000	500		15.3	5.1			250	.0204	
	10	M3		1000		30	10			500	.02	
	11	L3		2500		72	24			1250	.0192	
	12	G3		5000		141	47			2500	.0188	

Table 3: Accumulated Data for Benchmark I

Benchmark ID: I Benchmark Title: Unindexed Selection of 5% of Tuples

General Form of the Query: (cont'd)

Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ /RB	
I4	13	54	2000	500	3	30	10			500	.02	
	14	74		1000		60	20			1000	.02	
	15	L4		2500		141	47			2500	.0188	
	16	64		5000		282	94			5000	.0188	

Table 3: Accumulated Data for Benchmark I (Continued)

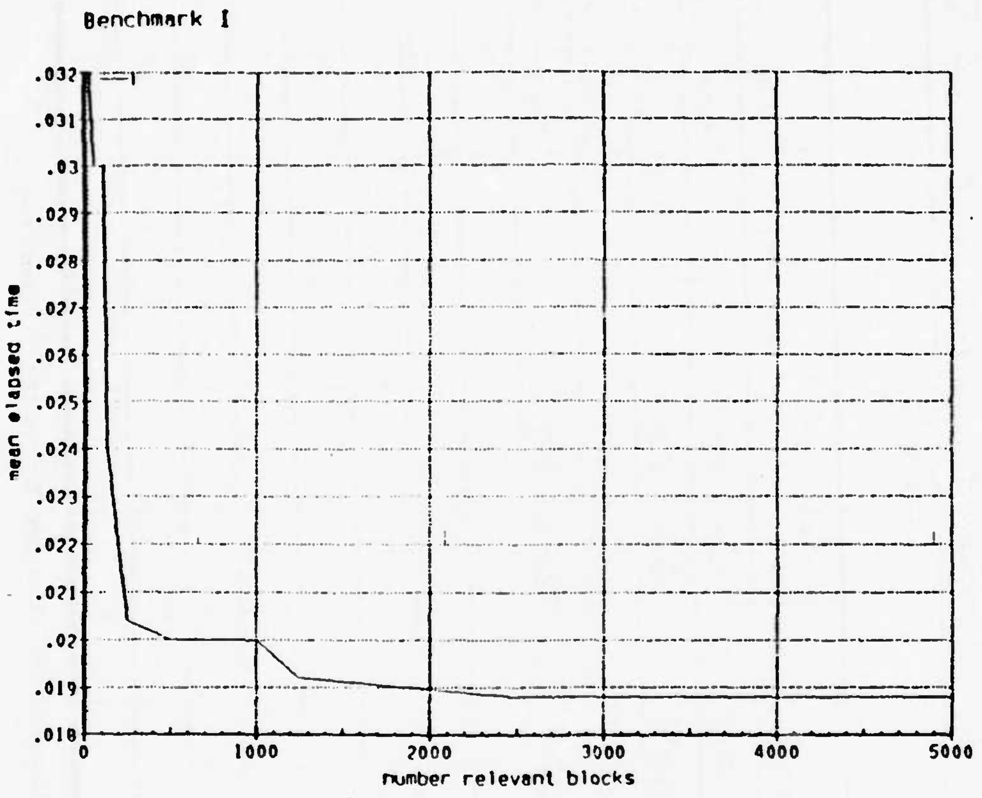


Figure 33: Mean Elapsed Times Per Relevant Block for Benchmark I

purposes in further benchmarks.

9.4. Benchmark II

This benchmark is titled "Selection of 5% of Tuples Qualified on Secondary-Key Attribute". The measurements for all sixteen relations in the database are shown in Table 4. Figure 34 shows the mean elapsed times per relevant block for this benchmark graphed against the means from Benchmark I. As the number of relevant blocks increases, the means from this benchmark approach the means from Benchmark I. This is the result of spreading the overhead of accessing and search the index over a larger number of blocks.

Figure 35 shows the mean elapsed time per relevant block curves for each tuple width, along with the means from Benchmark I for comparison. Figure 36 is the same data for tuple widths 3 and 4, with a larger scale on the y-axis for clarity. Figure 37 is a graph of the improvement ratio, showing the improvement factor afforded by qualifying the queries on a secondary key attribute. This graph shows that the improvement for the larger tuple widths approaches but does not reach the theoretical maximum of 20. With this evidence, we will classify the sixteen queries in this benchmark as overhead-intensive.

9.5. Benchmarks XII, XIII, and XIV

These are the projection benchmarks. In Benchmark XII, 25% of the attribute values are projected out of the tuples. In Benchmarks XIII and XIV, 50% and 75% of the attribute values, respectively, are projected. Except for the projection specified in the target list, these queries are identical to the queries of Benchmark II. The measurements are shown in Tables 5, 6, and 7. The mean elapsed times per relevant block for tuple widths 1, 2, 3, and 4 are shown in Figures 38, 39, 40, and 41, respectively. In every case, projection is more "expensive", i.e., takes more time, than retrieving the entire tuple. Is this because of the

Benchmark ID: II Benchmark Title: Select 5% of Tuples Qualified on S.K.A.

General Form of the Query: SELECT * FROM relation-name WHERE P5A = "value"

Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ /RB	
II 1	1	S1	100	500	3	1.53	.51	25	.0204	2	.255	
	2	M1		1000.		1.8	.6	50	.012	3	.2	
	3	L1		2500		2.4	.8	125	.0064	7	.11429	
	4	G1		5000		3.15	1.05	250	.0042	13	.08077	
II 2	5	S2	200	500		1.65	.55	25	.022	3	.183̄	
	6	M2		1000		1.68	.56	50	.0112	5	.112	
	7	L2		2500		2.4	0.8	125	.0064	13	.0615	
	8	G2		5000		3.15	1.05	250	.0042	25	.042	
II 3	9	S3	1000	500		2.4	0.8	25	.0032	13	.06154	
	10	M3		1000		2.85	.95	50	.019	25	.038	
	11	L3		2500		5.1	1.7	125	.0136	63	.02699	
	12	G3		5000		8.7	2.9	250	.0116	125	.0232	

Table 4: Accumulated Data for Benchmark II

Benchmark ID: II Benchmark Title: Select 5% of Tuples Qualified on S.K.A.

General Form of the Query: (cont'd)

Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ/RB	
II 4	13	S4	2000	500	3	3.15	1.05	25	.042	25	.042	
	14	M4		1000		4.5	1.5	50	.03	50	.03	
	15	L4		2500		8.4	2.8	125	.0224	125	.0224	
	16	G4		5000		15.78	5.26	250	.021	250	.02104	

Table 4: Accumulated Data for Benchmark II (Continued)

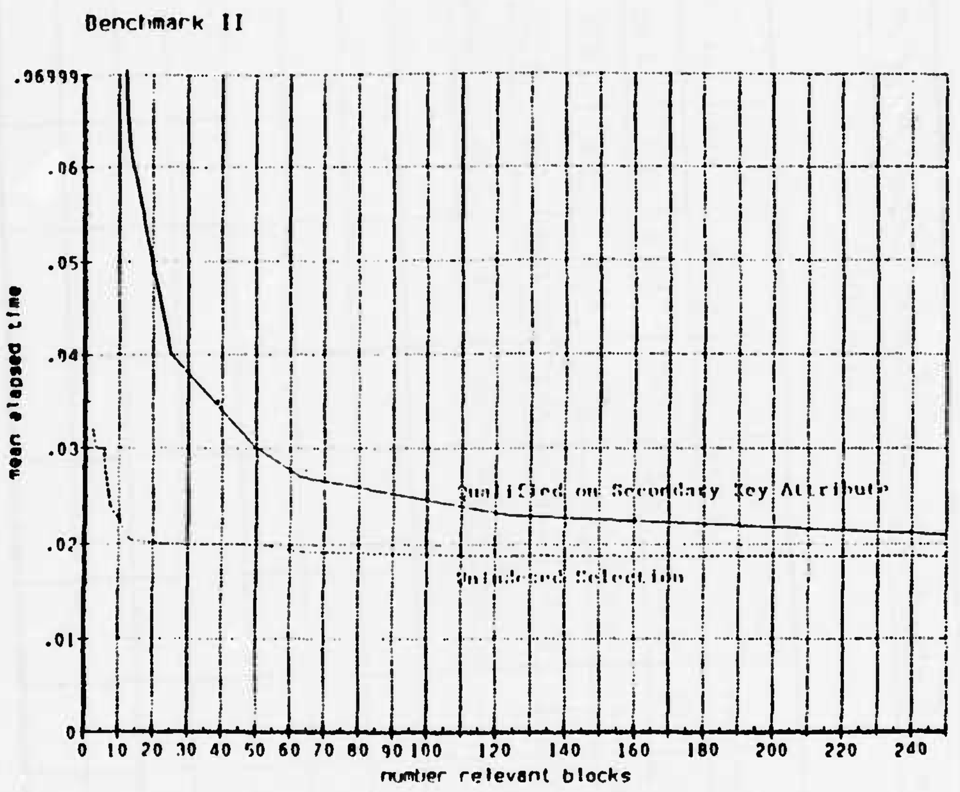


Figure 34: Mean Elapsed Times Per Relevant Block for Benchmarks I and II

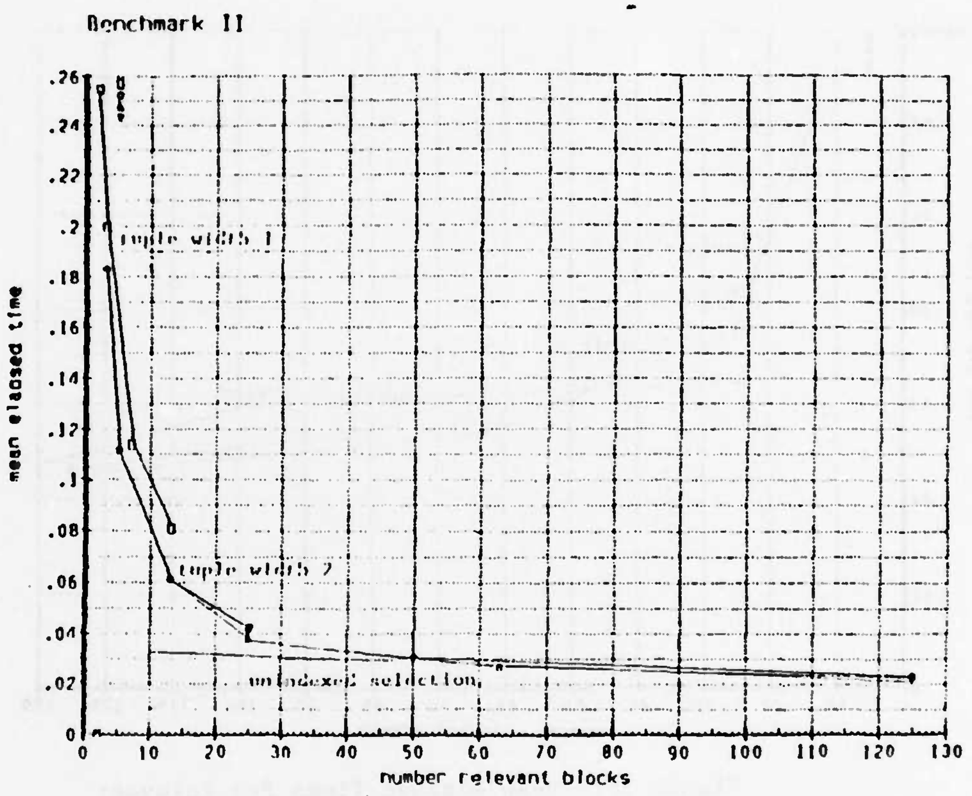


Figure 35: Mean Elapsed Times Per Relevant Block for All Tuple Widths

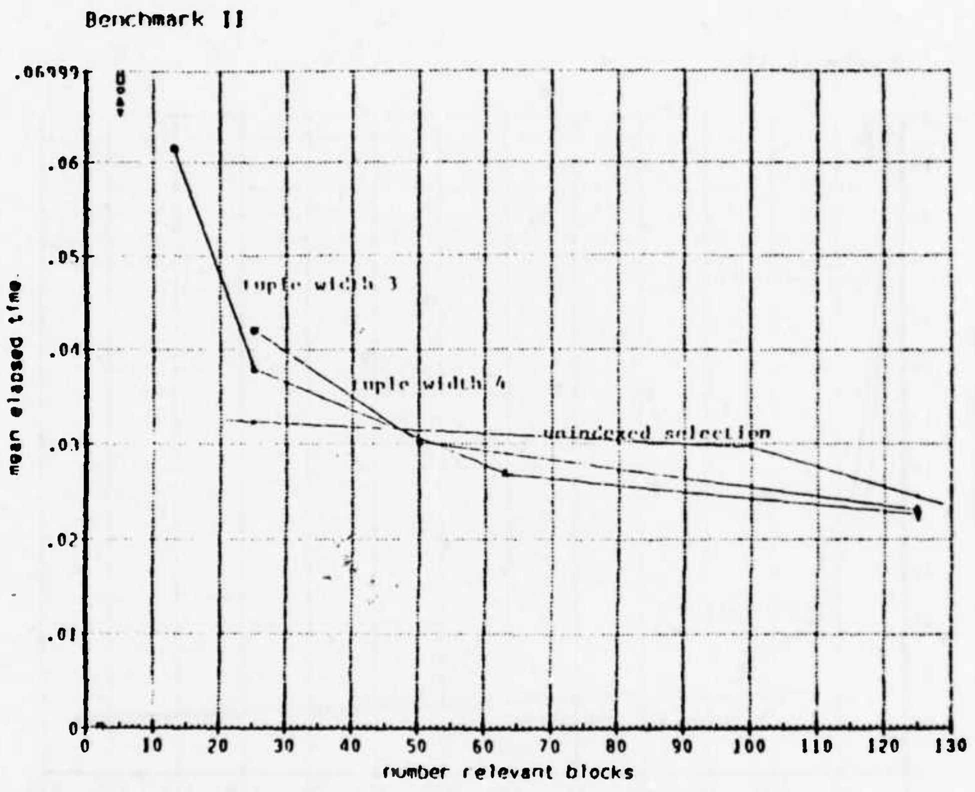


Figure 36: Mean Elapsed Times Per Relevant Block for Tuple Widths 3 and 4

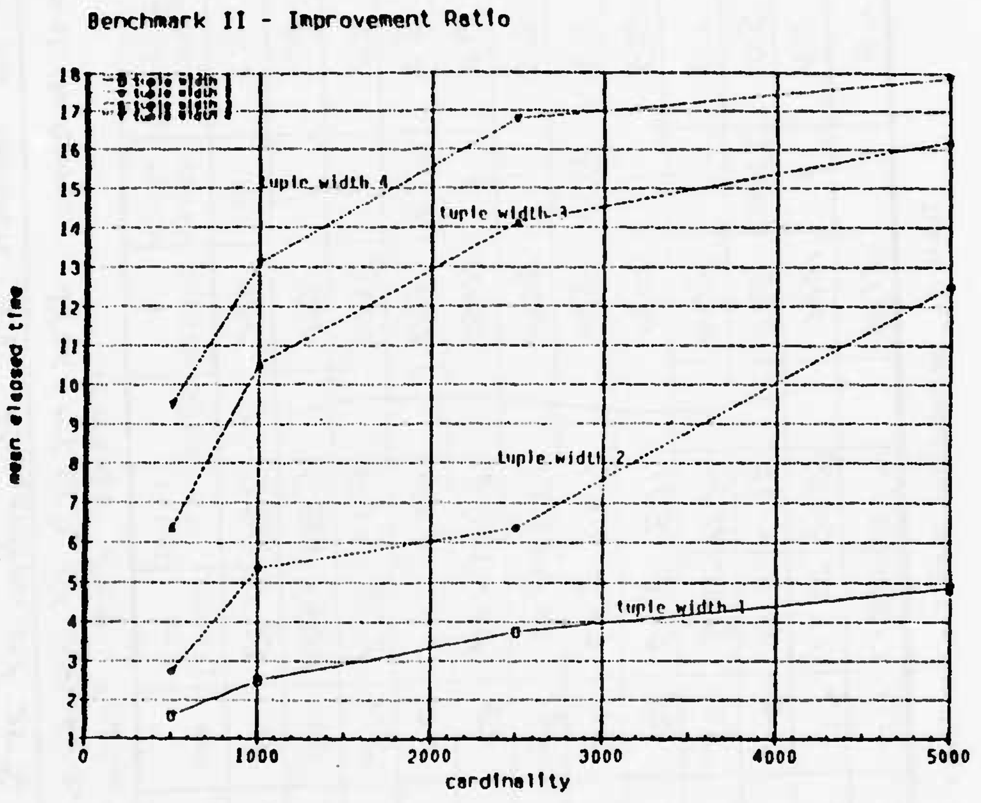


Figure 37: Improvement Ratio

Benchmark ID: XII	Benchmark Title: Projection of 25% of Attribute Values											
General Form of the Query:	SELECT (25% of attribute values) FROM relation name WHERE PSA = "value"											
Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ/RB	
XII 1	1	S1	100	500	3	3.983	1.33	25	.053	2	.664	
	2	M1	〃	1000	〃	5.983	1.994	50	.0399	3	.664	
	3	L1	〃	2500	〃	9.83	3.28	125	.026	7	.468	
XII 2	4	S2	200	500	〃	4.03	1.34	25	.054	3	.448	
	5	M2	〃	1000	〃	6.25	2.08	50	.042	5	.417	
	6	L2	〃	2500	〃	12.167	4.06	125	.033	13	.312	
XII 3	7	S3	1000	500	〃	6.216	2.07	25	.083	13	.159	
	8	M3	〃	1000	〃	11.3	3.77	50	.075	25	.151	
	9	L3	〃	2500	〃	75.58	25.9	125	.202	63	.400	
XII 4	10	S4	2000	500	〃	7.817	2.61	25	.104	25	.104	
	11	M4	〃	1000	〃	60.916	20.31	50	.406	50	.406	
	12	L4	〃	2500		147.517	49.17	125	.394	125	.394	

Table 5: Accumulated Data for Benchmark XII

Benchmark ID: XIII Benchmark Title: Projection of 50% of Attribute Values

General Form of the Query: SELECT (50% of attribute values) FROM relation-name WHERE PSA = "value"

Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ/RB
XIII 1	1	S1	100	500	3	5.183	1.73	25	.069	2	.864
	2	M1	〃	1000	〃	6.667	2.22	50	.045	3	.741
	3	L1	〃	2500	〃	12.967	4.32	125	.035	7	.618
XIII 2	4	S2	200	500	〃	7.567	2.52	25	.05	25	.841
	5	M2	〃	1000	〃	10.667	3.56	50	.071	50	.711
	6	L2	〃	2500	〃	22.067	7.36	125	.059	125	.567
XIII 3	7	S3	1000	500	〃	8.3	2.77	25	.111	25	.212
	8	M3	〃	1000	〃	15.25	5.08	50	.102	50	.203
	9	L3	〃	2500	〃	88.517	29.51	125	.236	125	.468
XIII 4	10	S4	2000	500	〃	11.683	3.89	25	.156	25	.156
	11	M4	〃	1000	〃	59.217	19.74	50	.395	50	.395
	12	L4	〃	2500	〃	156.5	52.17	125	.417	125	.417

Table 6: Accumulated Data for Benchmark XIII

Benchmark ID: XIV Benchmark Title: Projection of 75% of Attribute Values

General Form of the Query: SELECT (75% of attribute values) FROM relation-name WHERE PEA = "value"

Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ/RB	
XIV 1	1	S1	100	500	3	5.85	1.95	25	.078	2	.975	
	2	M1	〃	1000	〃	8.65	2.88	50	.058	3	.961	
	3	L1	〃	2500	〃	17.75	5.92	125	.047	7	.845	
XIV 2	4	S2	200	500	〃	9.633	3.21	25	.128	3	1.07	
	5	M2	〃	1000	〃	15.183	5.06	50	.101	5	1.01	
	6	L2	〃	2500	〃	30.583	10.19	125	.082	13	.784	
XIV 3	7	S3	1000	500	〃	11.083	3.69	25	.148	13	.284	
	8	M3	〃	1000	〃	18.617	6.21	50	.124	25	.248	
	9	L3	〃	2500	〃	94.517	31.51	125	.252	63	.500	
XIV 4	10	S4	2000	500	〃	17.7	5.9	25	.236	25	.236	
	11	M4	〃	1000	〃	76.367	25.46	50	.509	50	.509	
	12	L4	〃	2500		189.95	63.32	125	.507	125	.507	

Table 7: Accumulated Data for Benchmark XIV

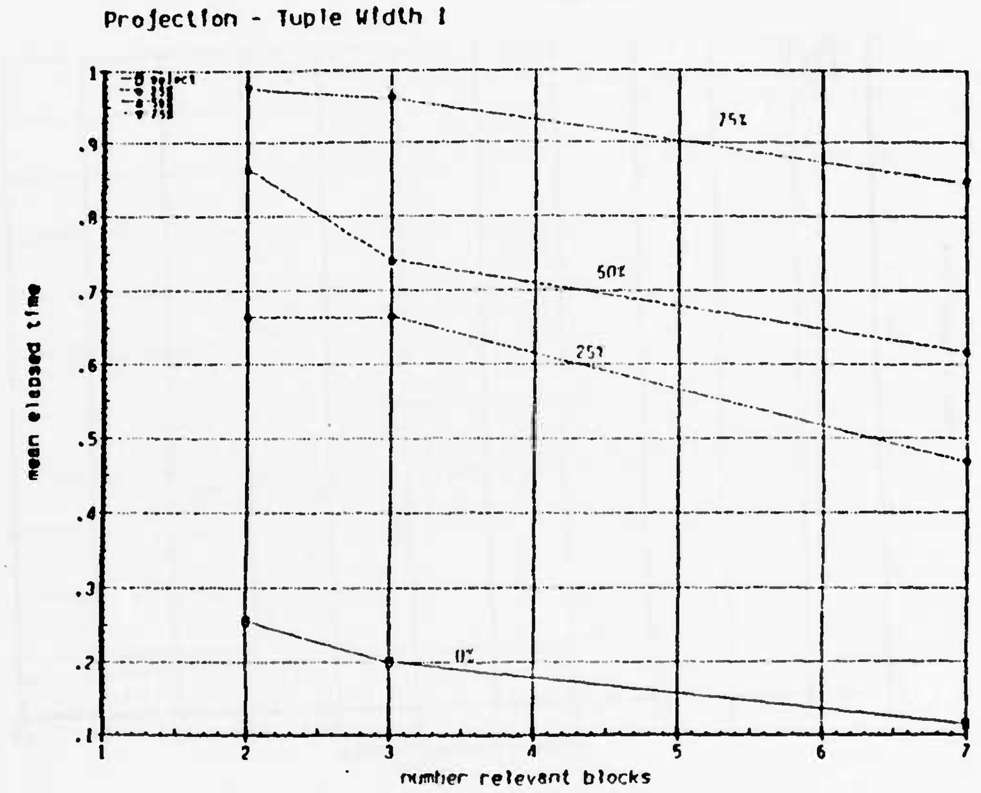


Figure 38: Projection for Tuple Width 1

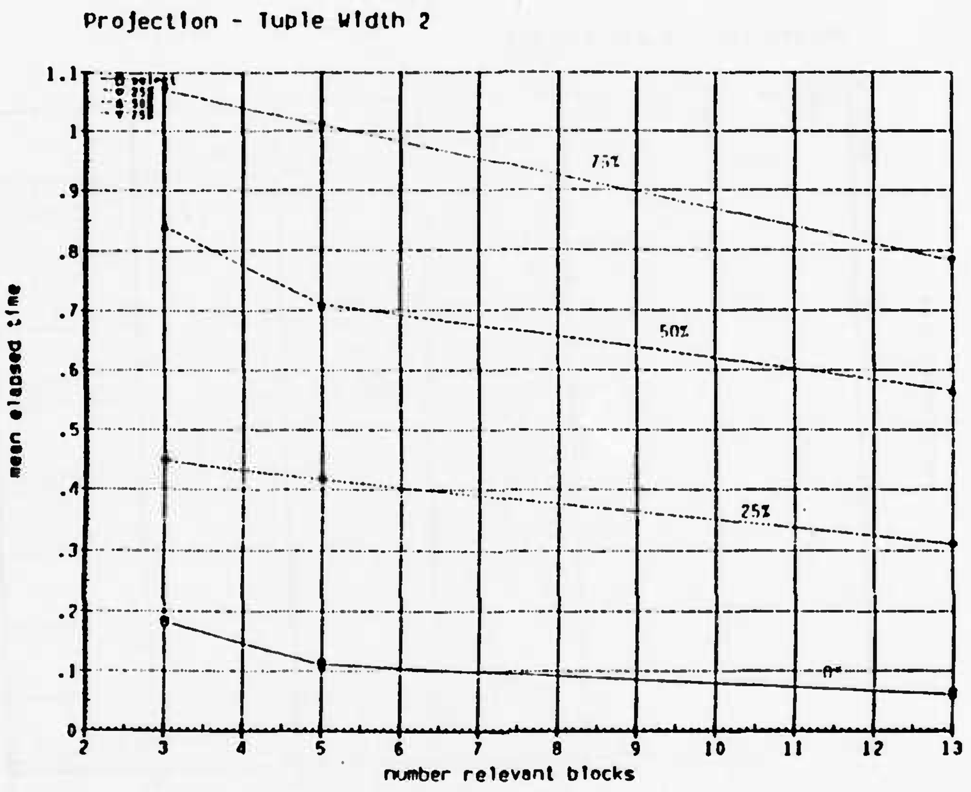


Figure 39: Projection for Tuple Width 2

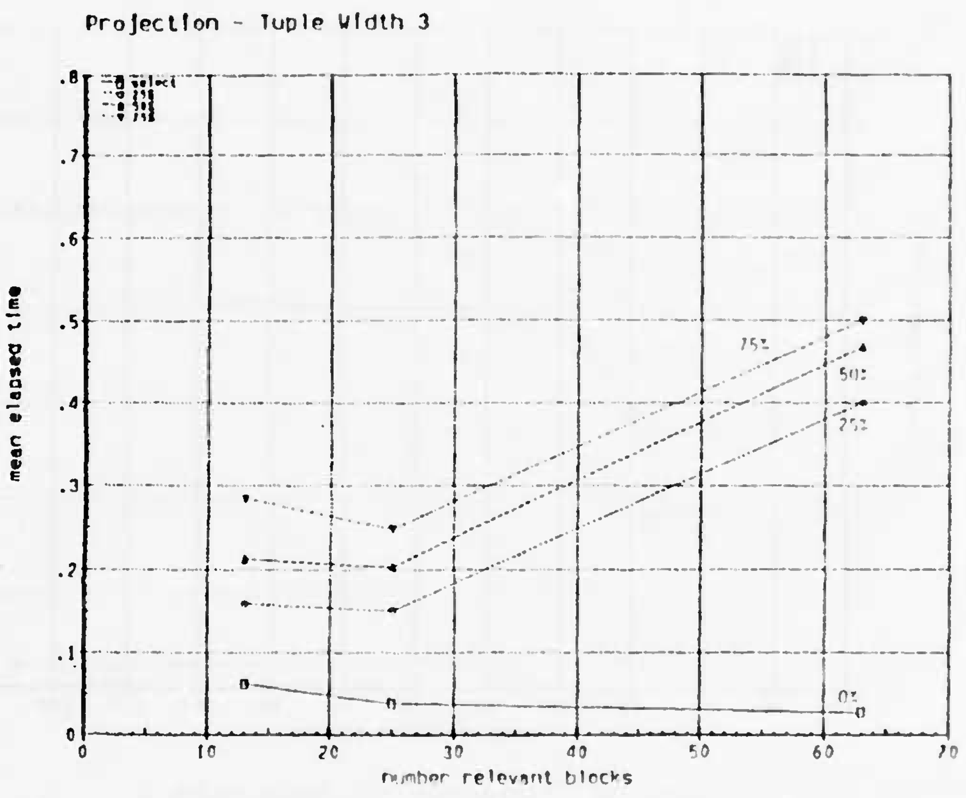


Figure 40: Projection for Tuple Width 3

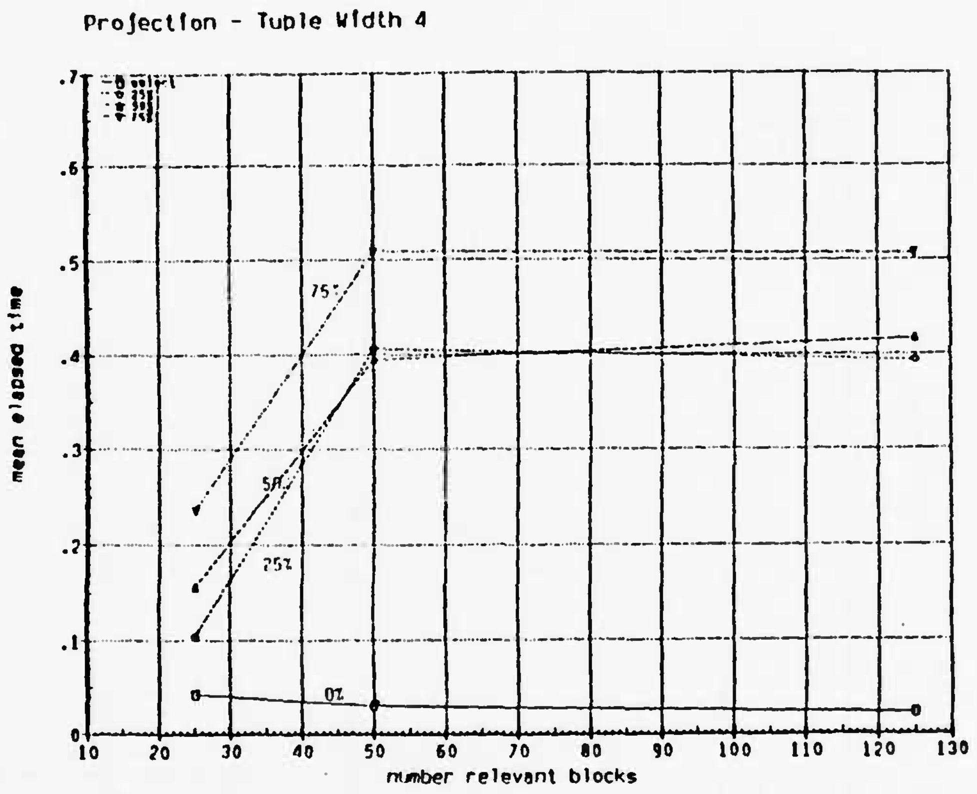


Figure 41: Projection for Tuple Width 4

additional work required by the processor, or it due to the limited size of the primary memory?

To answer this question, we graph the mean elapsed times per relevant block against the number of relevant blocks. This graph is shown in Figure 42.

Generally, the mean elapsed times per relevant block decrease with the number of relevant blocks up to 25 blocks. The means increase dramatically from 25 to 50 blocks, and then level out. This dramatic rise may be associated with some limitation in the way the main memory is allocated. The machine configuration has 2 M-bytes of primary memory. About 300 K-bytes are used by the system. This leaves 170 K-bytes, or about 85 blocks, for data. Some of the 85 blocks must be used to buffer output data. The rise in means at 50 blocks seems to indicate that the entire 85 blocks is not available for buffering data from the disk.

We will classify the 9th, 11th and 12th queries from each of Benchmarks XII, XIII, and XIV as access-intensive, based on the analysis of mean elapsed times per relevant block. The 1st through the 8th queries and the 10th query of each of these benchmarks, we will classify as processor-intensive.

9.6 Benchmark XV

This, the first of the join benchmarks, is titled "Equijoins on Non-Key Attribute". The experiments reported here include the following joins: S1 X M1, L1 X G1, S2 X M2, and L2 X G2. The measurements are given in Table 8. From these measurements, we will attempt to determine the join strategy being used. The smaller joins, S1 X M1 and S2 X M2, will require the same number of operations, since the cardinalities of the relations are equal. Likewise, the larger joins, L1 X G1 and L2 X G2, will require the same number of operations. The number of operations required for

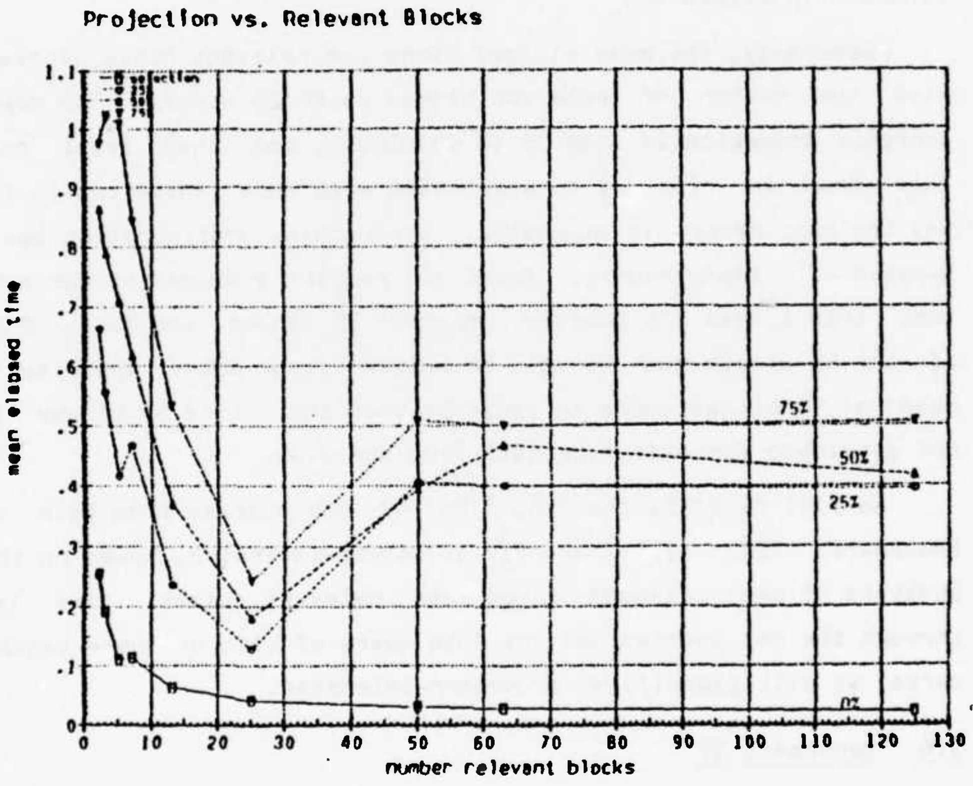


Figure 42: Mean Elapsed Times Per Relevant Block for Projection

Benchmark ID: XV Benchmark Title: Equijoin on Non-Key Attribute

General Form of the Query: SELECT (all attribute values from r1 and r2) FROM r1, r2 WHERE r1.COPY-KEY = r2.COPY-KEY

Job #	Q #	Relation Name	Tuple Width	# Tuples	Reps	Total ET	ET/ Query	r1+r2 #PT	μ/PT	#RB	μ/RB	
XV 1	1	S1XM1	100	500+ 1000	2	83.37	41.69	1500	.028	25+ 50	.556	
	2	L1XG1	100	2500+ 5000	2	1774.583	887.29	7500	.118	125+ 250	2.367	
XV 2	3	S2XM2	200	500+ 1000	1	540	54.0	1500	.036	50+ 100	.36	
	4	L2XG2	200	2500+ 5000	1	2152.3	2152.3	7500	.287	250+ 500	2.87	

Table 8: Accumulated Data for Benchmark XV

each join size for each algorithm is shown in Table 9.

Then, in Table 10, we show the mean elapsed time per operation computed for each of the joins and each of the algorithms. How can we determine which algorithm is being used? First, the mean elapsed times per participating tuple for S1 X M1 and L1 X G1 should be relatively close, since the number of tuples per block is the same and the primary memory is large enough to hold both relations. The means for the S2 X M2 and L2 X G2 joins should also be close. However, we might expect the means to rise from S2 X M2 to L2 X G2, since the latter involves 750 relevant blocks, close to the size of the available primary memory. The data in Table 10 seem to indicate that a straightforward join algorithm is being used. The variance in the measurements is an order of magnitude less than the variance for the next closest set of measurements. The mean elapsed times per operation for this algorithm are graphed in Figure 43.

Since the elapsed time per operation varies very little with the number of relevant blocks, and seems to be a function of the number of operations required, we classify the four queries from this benchmark as processor-intensive.

9.7. Benchmark XVI

The title of this benchmark is "Equijoin on Primary-Key Attribute". The measurements are shown in Table 11. Since the relations are joined on the primary key attribute, and the tuples are stored in order by primary key attribute, the most efficient join strategy would be a simple merge. Let us determine whether this strategy is being used.

First examine the mean elapsed times per relevant block graphed against tuple width on the x-axis in Figure 44. The mean elapsed times per relevant block are very close for joins having the same tuple widths, regardless of cardinality. This seems to indicate that performance is a function not of the number of

	Join Algorithm		
Join Size	Straightforward	Sort-and-Match	Hashing
Small	500,000	14,949	1500
Large	12,500,000	92,160	7500

Table 9: Number of Operations Required for Performing Join

	Join Algorithm		
Join	Straightforward	Sort-and-Match	Hashing
S1 X M1	0.00008	0.00279	0.028
L1 X G1	0.00007	0.00963	0.118
S2 X M2	0.00011	0.00361	0.036
L2 X G2	0.00017	0.02354	0.287

Table 10: Mean Elapsed Times Per Operation Computed for Three Join Algorithms

relevant blocks, but of the number of tuples per block.

However, as Figure 45 illustrates, the mean elapsed times per participating tuple increase for tuple widths 3 and 4. This might be explained by assuming that, although the number of comparisons per block decreases as tuple width increases, the work involved in formatting the larger tuples is affecting the means.

We assume that a merge algorithm is being used, and graph the mean elapsed times per operation, as shown in Figure 46. The shapes of the curves are the same as the shapes of the mean elapsed time per participating tuple curves. Since the mean elapsed times seem to be a function of the number of operations,

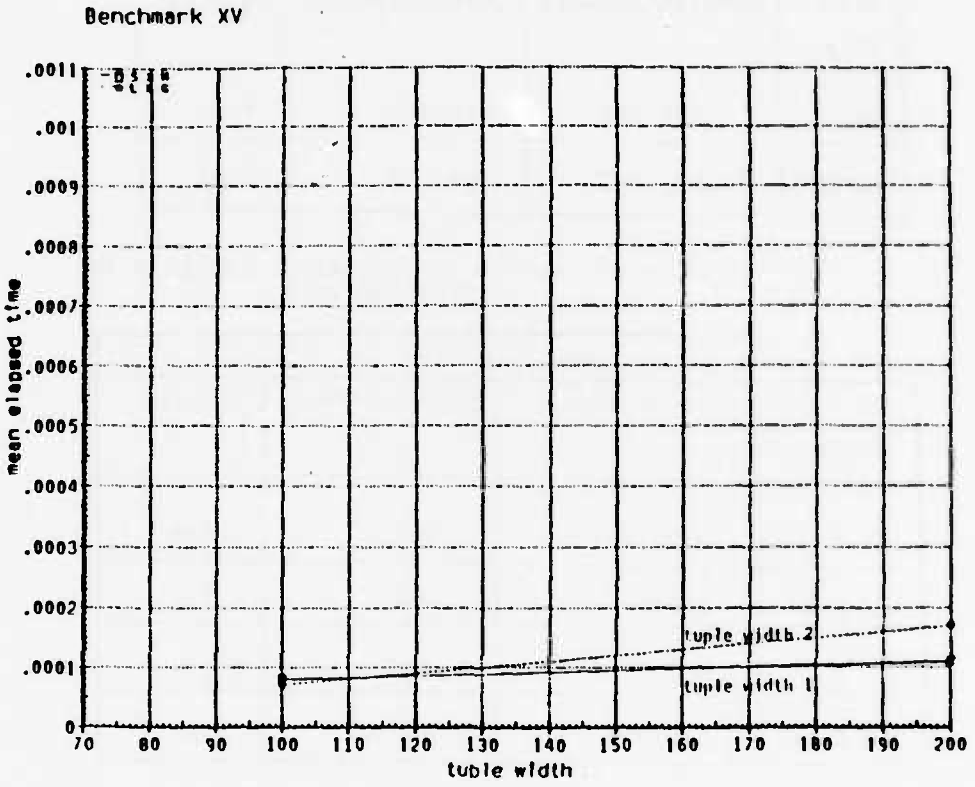


Figure 43: Mean Elapsed Times Per Operation for a Straightforward Join Algorithm

Benchmark ID: XVI Benchmark Title: EQUIJOIN ON PRIMARY KEY ATTRIBUTE

General Form of the Query: SELECT (all attribute values from r1, r2) FROM r1, r2 WHERE r1.KEY = r2.KEY

Job #	Q #	Relation Name	Tuple Width	r1+r2 # Tuples	Reps	Total ET	ET/ Query	r1+r2 #PT	μ/PT	#RB	μ/RB	
XVI 1	1	S1XM1	100	1500	3	128.5	42.8	1500	.0285	75	.571	
	2	L1XG1	∫	7500	3	635.03	211.68	7500	.0282	375	.564	
XVI 2	3	S2XM2	200	1500	3	132.88	44.29	1500	.0295	150	.295	
	4	L2XG2	∫	7500	3	645.57	215.19	7500	.0287	750	.287	
XVI 3	5	S3XM3	1000	1500	2	303.18	151.59	1500	.101	750	.202	
	6	L3XG3	∫	7500	2	1480.75	740.38	7500	.099	3750	.197	
XVI 4	7	S4XM4	2000	1500	2	500.87	250.44	1500	.167	1500	.167	
	8	L4XG4	∫	7500	2	2516.22	1258.11	7500	.168	7500	.168	

Table 11: Accumulated Data for Benchmark XVI

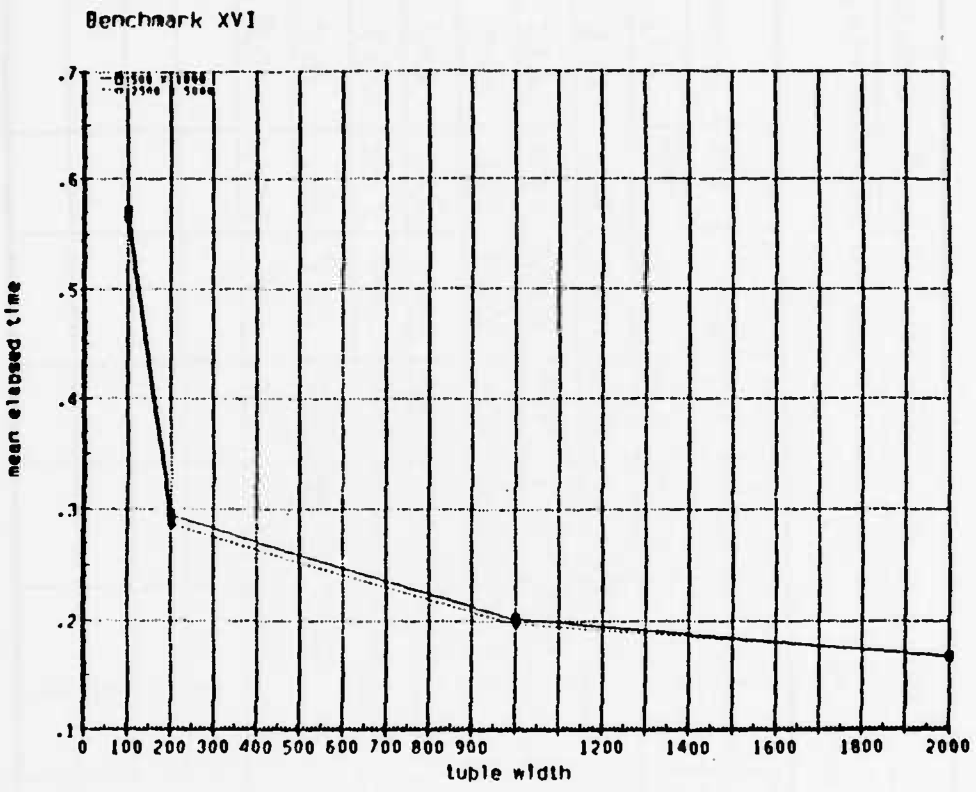


Figure 44: Mean Elapsed Times Per Relevant Block for Equijoins on Primary-Key Attribute

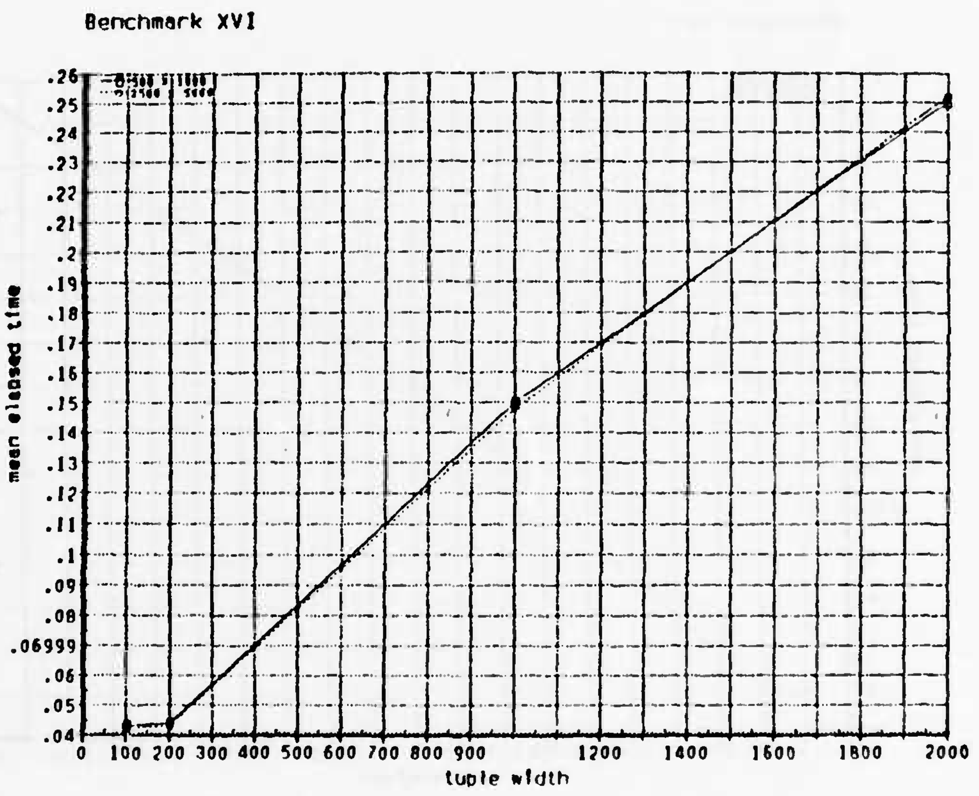


Figure 45: Mean Elapsed Times Per Participating Tuple for Equijoins on Primary-Key Attribute

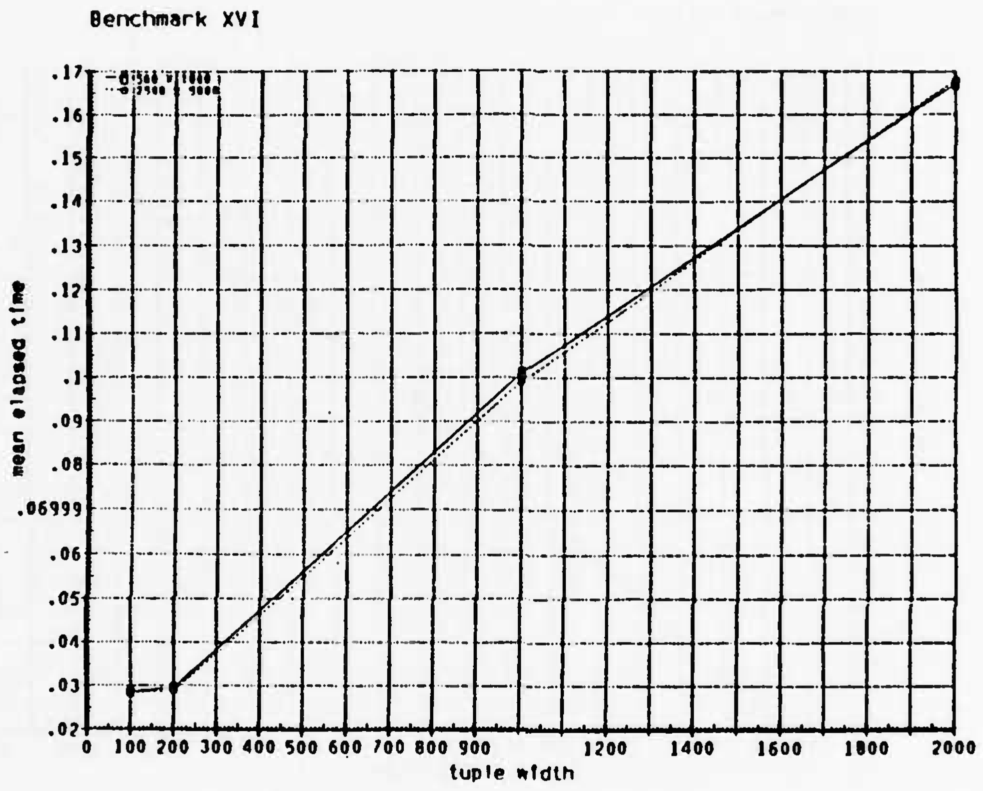


Figure 46: Mean Elapsed Times Per Operation
for Merge Algorithm

we classify these queries as processor-intensive.

9.8. Benchmark XVIII

The title of this benchmark is "Equijoin on Primary-Key Attribute with Join Predicate Reversed". We have two measurements, from the joins S1 X M1 and L1 X G1, for comparison with the measurements from Benchmark XVI. The measurements are shown in Table 12.

The mean elapsed times per operation, computed on the assumption that a merge algorithm is being used, are graphed in Figure 47, along with the means from Benchmark XVI. Although it is desirable to have more than two points for comparison, the results seem to indicate that reversing the join predicate has no adverse effect. Since the join involves no selection, the size of the relations can be determined from the dictionary or from the indices, and the smaller relation selected as the source relation.

9.9. Benchmark XIX

This benchmark is titled "Equijoin on Primary-Key Attribute with Selection on Secondary-Key Attribute". The queries in this benchmark have a selection predicate only on the source relation, not on both the source and target relations as specified for this benchmark. Two joins, S1 X M1 and L1 X G1, are performed, with the selection on the source relation limited to 5% of the tuples. The measurements are shown in Table 13.

We should consider three possible join algorithms. First, since selection is on secondary key, there is no guarantee that the tuples selected will be in order by primary key. Therefore a straightforward join algorithm might be used. However, it is likely that either the selected tuples will be sorted and then merged with the target relation, or that a semijoin strategy will be used. Table 14 shows the mean elapsed time per operation assuming a straightforward join algorithm and a sort-and-match algorithm, and a mean elapsed time per relevant block assuming the

Benchmark ID: XVIII Benchmark Title: Equijoin on P.K.A. w/Predicate Reversed

General Form of the Query: SELECT (all attribute values of r1 and r2) FROM r1, r2 WHERE r2.KEY = r1.KEY

Job #	Q #	Relation Name	Tuple Width	r1+r2 # Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#ops ~~#RB~~	μ/op ~~μ #RB~~	
XVIII 1	1	S1XM1	100	1500	1	43.386	—			1000	.043	
	2	L1XG1	∫	7500	1	199.398	—			5000	.040	
			#ops =	2(C(r1)), C(R1) ≤ C(R2)								

Table 12: Accumulated Data for Benchmark XVIII

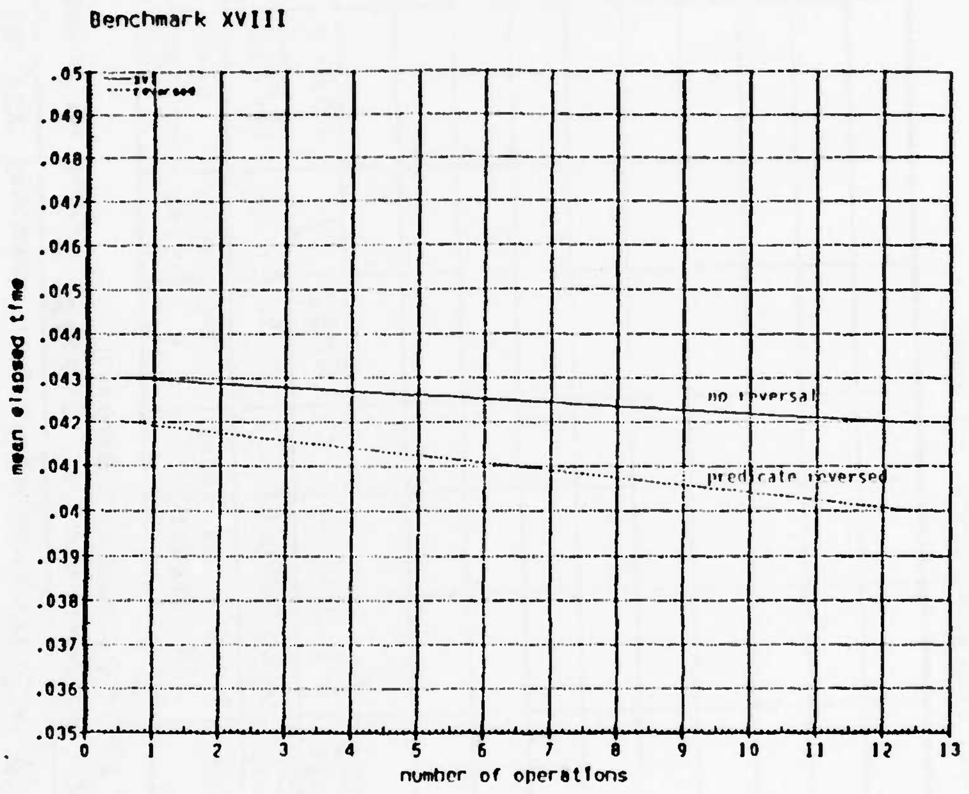


Figure 47: Equijoin on Primary-Key Attribute
with Join Predicate Reversed

Benchmark ID: XIX Benchmark Title: Equijoin on P.K.A w/selection on S.K.A.

General Form of the Query: SELECT (all attributes of r1 and r2) FROM r1,r2.
WHERE r1.P5A = "value" AND r1.KEY=r2.KEY

Job #	Q #	Relation Name	Tuple Width	r1+r2 # Tuples	Reps	Total ET	ET/ Query	#PT	μ/PT	#RB	μ /RB	
XIX 1	1	S1XM1	100	1500	2	3.908	1.954	25+ 1000	.00191	52	.03758	
	2	L1XG1	ʃ	7500	2	12.658	6.329	125+ 5000	.00123	257	.02463	

Table 13: Accumulated Data for Benchmark XIX

	Join Algorithm		
Join	Straightforward	Sort-and-Match	Semijoin
S1 X M1	0.000078	0.00177	0.4885
L1 X G1	0.00001	0.00105	0.4521

Table 14: Means for Sort-and-Match and Semijoin Algorithms for Equijoin with Selection

semijoin algorithm.

Although our sample of two joins is too small for definitive pronouncements, it seems likely that either a semijoin or a sort-and-match strategy is being used. In both cases the related means are relatively close. The means for the semijoin are proportionately closer than the means for the sort-and-match. Let us assume that the semijoin algorithm is being used, and classify the queries as overhead-intensive.

9.10. The Performance Indices

Although we have not run all the basic benchmarks, we have collected enough data to illustrate the computation of the performance indices. We will compute $\overline{EA}$, $\overline{EP}$, and $\overline{EO}$. These indices represent the best-case performance of the machine for the access-intensive, processor-intensive, and overhead-intensive classes of queries. These indices will be used to determine the composition of the standard benchmark jobs for multiple-user benchmarks, as illustrated in the next section of this chapter. These indices are also the basis for constructing the $\overline{EA}'$, $\overline{EP}'$, $\overline{EO}'$, and $\overline{E}'$ indices for multiple-use benchmarks.

First we calculate $\overline{EA}$, the performance index for access-intensive queries. The access-intensive query pool contains a total of 25 queries. These are the sixteen queries from Benchmark I, and the 9th, 11th, and 12th queries from each of benchmarks XII, XIII, and XIV. The mean of the mean elapsed times per relevant block, calculated from the means for these queries shown in tables 3, 5, 6, and 7, is 0.174. The performance index $\overline{EA}$, is then 0.174/20, since the largest number of tuples per block in the database is 20. So $\overline{EA}$ = 0.0087.

Now we calculate $\overline{EP}$, the performance index for processor-intensive queries. The processor-intensive query pool contains a total of 39 queries. These are the first through the eighth queries from each of benchmarks XII, XIII, and XIV, the 10th query from each of the same benchmarks, and all of the equijoin queries from benchmarks XV and XVI. The performance index $\overline{EP}$ can be computed by taking the mean of the mean elapsed times per participating tuple for these queries. Using the data from tables 5, 6, 7, 8, and 11, we calculate $\overline{EP}$ = 0.0854.

Finally, we calculate $\overline{EO}$, the performance index for overhead-intensive queries. The overhead-intensive query pool contains a total of 18 queries. These are the sixteen from Benchmark II, and the two semijoin queries from Benchmark XIX. $\overline{EO}$ is the mean of the mean elapsed times per participating tuple of these queries. Using the data from Tables 4 and 13, we calculate $\overline{EO}$ = 0.174.

9.11. Constructing Standard Jobs

Using the data from the benchmarks reported in the previous sections and the performance indices calculated in the preceeding section, let us construct standard jobs for multiple-user benchmarks. First we will construct the standard access-intensive job, then the standard processor-intensive job, and finally the standard overhead-intensive job.

We will construct the jobs as follows. First let us take the mean elapsed times from a query pool, and arrange them in order by mean elapsed time. Then we will inspect the data for clustering tendencies. In the examples presented here, we have clustered the means by inspection since the number of means is small. (Appendix F contains the results of a cluster analysis analysis of the data, with the results compared to the clusters arrived at by inspection.)

Next, we will select from the clusters of means the smallest set of clusters such that the mean of the cluster means is very close to the performance index for the query pool. Then from the queries in the selected clusters, we will select a small set of queries such that the mean of the query means is very close to the performance index. This set of queries will be the set of queries for the standard job.

Once all the sets of queries for the standard jobs have been selected, we will examine the mean elapsed times per query in each query pool to determine total job elapsed time, and the elapsed time per query for the standard jobs.

First we determine the composition of the standard access-intensive job. The access-intensive query pool includes the sixteen queries from Benchmark I, and queries 9, 11, and 12 from each of benchmarks XII, XIII, and XIV. Figure 48 shows the mean elapsed times per relevant block for the access-intensive query pools. The means have been abstracted from tables 3, 5, 6, and 7. Figure 48 shows these means ordered and divided into clusters. The clusters were determined by inspecting the means. Means which are very close are assigned to the same cluster. Each cluster is assigned a unique identifier. There are a total of six clusters. We want to select a small set of clusters such that the mean of the cluster means is close to $\overline{EA}$, which is 0.174. The mean of the cluster means of clusters 2, 3, and 5 is 0.174. So we will select the queries for the standard access-intensive job from the queries in clusters 2, 3, and 5. We want to select a small set of queries such that the mean of the query means is very close to $\overline{EA}$. So, we select from those clusters queries having mean elapsed times of 0.024, 0.03, and 0.468. The mean of these means is 0.174. These queries, then, will make up our standard job. By inspecting tables 3, 5, 6, and 7, we can identify these queries.

```
0.0188                     0.394
0.0188                     0.395
0.0188                     0.4      #4
0.0192                     0.4      mean=0.4012
0.02    #1                 0.417
0.02    mean=0.0197
0.02                       0.468    #5
0.02                                mean=0.468
0.0204                     0.500
0.0204                     0.507    #6
0.0204                     0.509    mean=0.505

0.024   #2
        mean=0.024

0.03
0.03    #3
0.03    mean=0.0305
0.032
```

Figure 48: Clusters for the
Access-Intensive Query Pool

Table 15 shows for each query the benchmark, the query number, the mean elapsed time per relevant block, and the elapsed time per query.

Now we go through the same process to determine the composition of the standard processor-intensive job. The processor-intensive query pool contains the first through eighth queries from each of benchmarks XII, XIII, and XIV, the 10th query from each of the same benchmarks, and all of the equijoin queries from benchmarks XV and XVI. Figure 49 shows the mean elapsed times per participating tuple for these queries, abstracted from tables 5,

Benchmark	Query#	Mean ET	ET/Query
I	3	0.024	3
I	2	0.03	1.5
XIII	9	0.468	29.51

Table 15: Queries for the Standard
Access-Intensive Job

6, 7, 8, and 11. The means have been ordered, and the clusters have been assigned by inspection. $\overline{EP}$ is 0.0854. If we select clusters 2, 4, 5, and 6, then the mean of the cluster means is 0.08427, approximating $\overline{EP}$. If we than select from these clusters the queries with means of 0.0399, 0.083, 0.101, and 0.118, the mean of the query means is 0.08547. This is very close to $\overline{EP}$. Therefore the four queries with means as specified above will comprise the standard processor-intensive job. By inspecting tables 5, 6, 7, 8, and 11, we can identify these queries. Table 16 shows for each query the benchmark and query number, the mean elapsed time per participating tuple, and the mean elapsed time per query.

Finally, we determine the composition of the standard overhead-intensive job. The overhead-intensive query pool contains all of the queries from benchmarks II and XIX. Figure 50 shows the mean elapsed times per participating tuple, abstracted

```
0.028               0.069
0.0282  #1          0.071   #4
0.0285  mean=0.0286 0.075   mean=0.0763
0.0287              0.078
0.0295              0.082
                    0.083
0.033
0.035   #2
0.036   mean=0.036  0.099
0.0399              0.101   #5
                    0.102   mean=0.1015
0.045               0.104
0.047
0.050   #3          0.118   #6
0.053   mean=0.0523 0.124   mean=0.1233
0.054               0.128
0.058
0.059               0.148
                    0.156   #7
                    0.167   mean=0.1598
                    0.168

                    0.236   #8
                    0.287   mean=0.2615
```

Figure 49: Clusters for the Processor-Intensive Query Pool

Benchmark	Query#	Mean ET	ET/Query
XII	2	0.0399	1.994
XII	7	0.083	2.07
XIV	5	0.101	5.06
XV	2	0.118	887.29

Table 16: Queries for the Standard Processor-Intensive Job

from tables 4 and 13. The means have been ordered and divided into clusters by inspection. The value of $\overline{EO}$ is 0.174. If we select clusters 2 and 3, then the mean of the cluster means is 0.01745, approximating $\overline{EO}$. If we select from those clusters queries with means of 0.025, 0.0136, and 0.0136, the mean of the query means is 0.174. These queries will comprise the standard overhead-intensive job. By examining tables 4 and 13, we can identify the queries. Table 17 shows for each query the benchmark, query number, mean elapsed time per participating tuple, and mean elapsed time per query.

```
0.0032  #1              0.03    #4
0.0042  mean=0.0049     0.039   mean=0.0405
0.0064                  0.042

0.0112
0.0116  #2
0.012   mean=0.0121
0.0136

0.019
0.0204
0.021   #3
0.022   mean=0.0228
0.0224
0.025
```

Figure 50: Clusters for the Overhead-Intensive Query Pool

Benchmark	Query#	Mean ET	ET/Query
II	11	0.0136	1.7
II	11	0.0136	1.7
XIX	2	0.025	6.329

Table 17: Queries for the Standard Overhead-Intensive Job

Now we must determine the job length. Four queries were chosen for the processor-intensive job. The mean elapsed time per query for the processor-intensive queries, calculated from the mean elapsed times per query in Table 14, is 228.107. Clearly this will result in an unacceptable job run time. Let us arbitrarily limit the elapsed time per query to 30 minutes. Then the will have a total job elapsed time is 120 minutes, the product of this elapsed time per query and the number of queries in the standard processor-intensive job. Each query in the processor-intensive job will run for 30 minutes. Each query in the access-intensive job will run for 120/3 = 30 minutes. Each query in the overhead-intensive job will run for 120/3 = 30 minutes.

So we have determined the composition of the standard jobs for multiple-user benchmarks, the total job elapsed time for the standard jobs, and the total elapsed time per query for each standard job. The next step in applying the methodology would be to construct the standard jobs, and run the standard job mixes specified in Chapter 8.

10. CONCLUSION

In the first nine chapters of this thesis, we have presented a methodology for benchmarking relational database machines. In this, the final chapter of the thesis, we will evaluate the work. The characteristics of the workload model, as defined in Chapter 2, form a framework for the evaluation. First, we identify the contributions of this work to the body of knowledge of database machines, database systems, and performance evaluation. Then, we discuss the directions for further research which arise from this thesis. The contributions of this work are discussed in Section 10.1. Some directions for further research are identified in Section 10.2.

10.1. Contributions of the Work

The methodology for benchmarking relational database machines is the first work of its kind. It comes at a time when database machines are moving from pencil-and-paper designs into viable, commercial implementations. The establishment of a standard for evaluating the performance of these machines at their emergence is of great value. At present, there is no established standard. In the literature, some give performance statistics in terms of "transactions per second". Ad hoc 'benchmarks' are run with an arbitrary choice of database structure and a severely limited number of transaction types. These are not methodologies, and do not furnish a standard for comparison.

First let us review the specific contributions of the methodology. Then let us examine some more general contributions.

10.1.1. The Specific Contributions of the Methodology

This work is the first step toward providing a standard for benchmarking relational database machines and relational software database management systems. A methodology, a collection of methods, tools, and procedures, is proposed. That a methodology is proposed is significant. Thus we are assured that the managerial as well as the technical aspects are addressed. We are assured that the benchmarks will be constructed and executed in a uniform fashion.

Let us review the composition of the methodology. The methodology is based on a three-level hierarchy of models. At the lowest level is the machine model. The machine is modeled as a high-level language architecture, i.e., as a machine which executes a relational query language. This approach gives a degree of machine-independence. The next level in the hierarchy is the database model. The database is modeled as a synthetic database consisting of twelve relations of varying tuple width and cardinality. The relations are based on a standard tuple template. This approach gives us database-independence, i.e., the database model is independent of any real-world database. The highest level in the hierarchy is the applications model. We present two applications models.

The first applications model consists of a collection of basic benchmarks. The idea of the basic benchmarks is to measure the best-case performance of the machine. The methodology includes a method for constructing the basic benchmark jobs, a procedure for executing the basic benchmark jobs, detailed specification of the benchmarks to be executed, and methods for interpreting the measurements from the benchmarks.

The basic benchmark jobs are constructed using one of two simple job models. The benchmarks are executed in single-user mode, in a carefully controlled environment. Detailed specifications are given for twenty-five basic benchmarks. There are seven benchmarks which measure machine performance for selection operations, five which measure performance for ordering and aggregation, three which concern projection, nine which measure performance for a variety of join operations, and two which measure the performance of the machine for update, delete, and insert operations.

A general procedure for analyzing the measurements is supplemented by the detailed instructions given with each benchmark specification. The analysis of the measurements results in a classification of the benchmark queries. The classification scheme is based on the resources used in execution. There are three classes of queries: access-intensive, processor-intensive, and overhead-intensive. Three simple performance indices, one for each classification, are developed to express the performance of the machine for each of these query classes.

The classification of queries and the performance indices from the basic benchmarks are used to develop the second applications model. The second applications model is a multiple-user benchmark model. The idea of the multiple-user benchmarks is to compare the performance of the machine with multiple users to the best-case performance measured in the basic benchmarks. The data for each query classification collected from the basic benchmarks and the performance indices from the basic benchmarks are used to construct standard jobs for multiple-user benchmarks. A standard access-intensive job, a standard processor-intensive job, and a standard overhead-intensive job are constructed. Performance indices which reflect the actual performance of the machine with multiple users are developed. In addition, a set of comparative indices are developed to reflect actual vs. ideal performance.

Experiments in an application of the methodology are reported. Nine of the basic benchmarks are included in the experiments. The results show that the methodology provides useful information about the performance of the benchmarked machine. The calculation of the performance indices and the construction of standard jobs for multiple-user benchmarks are illustrated using the measurements collected in these experiments.

10.1.2. The General Contributions of the Work

We have defined the goal of this thesis as the development of a methodology for benchmarking relational database machines for comparative evaluation of alternative machine architectures and machine configurations. But what is perhaps the most interesting contribution of the work developed along the way to that goal. During the process of actual experimentation, we discovered that we could learn more than merely how fast the machine performs certain tasks. The experiments show that, with a general knowledge of the machine architecture, we can interpret measurements taken from <u>outside</u> the machine to give us a great deal of information about what goes on <u>inside</u> the machine. This information is very useful to potential users of the machine. The experiments identify certain performance characteristics which should be considered in database design and applications development. This information, as well as the performance indices, should be considered when evaluating alternative machine architectures.

The work also contributes to the development of a standard for evaluating the performance of relational database machines. Let us discuss this in terms of three of the workload model characteristics defined in Chapter 2: generality, reproducibility, system-independence and comparability. <u>Generality</u> is the property that the benchmarks are applicable for a wide variety of databases and applications, and that the results are valid across differing database machine architectures. <u>Reproducibility</u> is the property that repetition of the experiments produces consistent results. <u>System-independence</u> is the property that the benchmarks can be transported from system to system, while still remaining representative. <u>Comparability</u> is the property that the benchmark results can be compared from system to system.

Our methodology incorporates these characteristics in the workload model. The methodology is based on a machine-independent machine model, a database-independent database model, and an application-independent applications model. The independence of these models gives us generality and system-independence. The methodology also specifies the environment in which the benchmarks are to be executed. By defining the host workload model, the benchmark job models, and the procedures for executing the benchmarks, we achieve reproducibility. The methodology also includes precise definitions for the performance metrics, or performance indices. These definitions, together with generality, system-independence, and reproducibility, give us comparability.

A third important contribution has to do with proving the price/performance advantage of database machines. We believe that the methodology can be extended with few modifications to software relational database management systems (DBMSs). Throughout the thesis, we have discussed relational database machines. However, the machine model is a high level model, specified at the level of the relational query languages. Such a model can be applied to software relational DBMSs as well as to relational database machines. Also, we note that evaluation is based on the classification of queries into the access-intensive, processor-intensive, and overhead-intensive categories. We expect that these same categories will hold for software DBMSs. Machine resources in both cases are processor and memory. Also, software DBMSs incur the same kinds of overhead as do database machines. Therefore, we may use the methodology to evaluate <u>relational database systems</u>, i.e., software DBMSs as well as database machines.

As a fourth contribution, the benchmark results will be useful in the area of modeling relational database system performance. Empirical measurements of performance can be used to develop and tune DBMS and database machine models.

10.2. Directions for Further Research

We begin our discussion of directions for further research by examining the remaining characteristics of workload models. In the previous section, we discussed generality, reproducibility, system-independence, and comparability. In this section we will examine representativeness, flexibility, simplicity of construction, compatibility, compactness, and usage costs.

Representativeness is defined as the accuracy of the model. We present three questions about representativeness which lead us to further research. First, how representative of real applications is our set of benchmarks? We constructed the benchmarks using the concept of the database system as a high-level language computer. The benchmarks are based on the operations of the relational query language. This assures us some degree of representativeness. But is our set of basic benchmarks sufficient? One issue which we have not addressed is that of arbitrarily complex queries. Is this important in real applications?

These questions lead us to one direction for further research, a survey of existing and proposed applications, with some classification and characterization of types of applications as a result. For example, a classification might be based on a spectrum of applications from static to highly volatile. Banking systems and airline reservation systems would be near the volatile end of the spectrum. General business applications would fall somewhere toward the static end of the spectrum. Whether or not this kind of classification scheme or another is appropriate can only be determined through further research.

Second, what can we do to better model concurrency in multiple-user benchmarks? The jobs in our multiple-user benchmark model are constructed without regard to concurrency. The probabilities of conflict, contention, and sharing are computed, but the computation is made after the composition of the jobs has been determined. Is there a way to construct multiple-user benchmarks with specified levels of conflict, contention, and sharing?

A third question concerning representativeness arises from the fact that backend database machines are designed with the idea of supporting multiple, dissimilar hosts. What effect does the support for multiple databases accessed from multiple, dissimilar hosts have on performance?

Next we examine <u>flexibility</u>, the capability to easily modify the model to reflect variations in real work load. We have provided flexibility in constructing the mix of jobs for multiple-user benchmarks. We might consider ways of modeling these jobs which allow more flexibility. Would it be desirable to develop different standard job models for different classifications of applications? Before we can do this, we must identify the classes of applications.

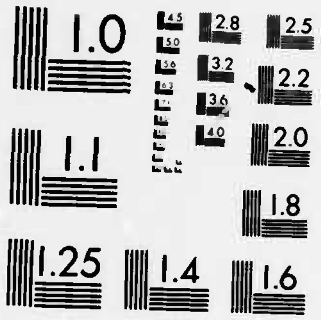
<u>Simplicity of construction</u>, the characteristic which we examine next, includes the cost and complexity of gathering the information required to design the workload model and to make it operational. The parameters for designing the workload model are carefully defined in our methodology, so that the complexity of gathering information is minimized. The cost of making the model operational includes the cost of constructing the database and the cost of developing the benchmark jobs. The cost of constructing the database will include the cost of developing the relation generator. It is possible to write the benchmark jobs and the relation generator in a programming language which is portable. This will reduce the costs of development to those required for modifying existing software. We see no new research directions here.

However, in the area of <u>compatibility</u> of the model and the system, we see two issues. The first, a low-level issue, concerns the compatibility of the query language between relational systems. We have specified the benchmark queries in SQL, which is the most widely used relational query language. These benchmarks will be compatible where an SQL interface is available.

AD-A139 104 A METHODOLOGY FOR BENCHMARKING RELATIONAL DATABASE MACHINES(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P R STRAWSER JAN 84 NPS52-84-004 3/3

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
6-84
DTIC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

Translation from SQL to other relational query languages will be required for some benchmarks. Most relational query languages have origins in the relational algebra or the tuple calculus. That the two are equivalent in power has been demonstrated. Therefore, we do not feel that translation is an direction for further research.

The second compatibility issue is a high-level issue. The methodology is targeted for relational database machines. As we indicated in the previous section, we believe that the methodology can be extended to relational software DBMSs with little modification. However, this has yet to be done. More interesting is the idea of generalizing the methodology to other models of data. This implies that we must develop a general model of the database structures and of the data management operations. Some preliminary work in this direction, using the attribute-based model of data as the general model, has been published in [Bann78a], [Bann78b], and [Bann80]. This may provide a basis for the development of a general methodology for benchmarking database systems.

Usage costs and compactness, the degree of detail of the model, are the final characteristics which we examine. The model presented in our methodology consists of 25 basic benchmarks, with a total of 112 queries and 15 query streams consisting of 32 queries each. If the average run time for a query or query stream is 15 minutes, this represents 1805 minutes or more than 30 hours run time for executing the basic benchmarks. With three repetition for each basic benchmark, the run time required is 3 * 30 or 90 hours, nearly four days. Additional time will be required for the multiple-user benchmarks. Thus the methodology, evaluated by itself, is neither particularly compact nor inexpensive to use. But consider that once the standard benchmarks have been run with a particular host-database-machine configuration, they need not be repeated. Therefore the cost of executing the standard benchmarks is a one-time cost for a particular configuration. Even taking this into consideration, whether we can reduce the standard benchmark set is an issue worthy of further research. While restricting the set of benchmarks is desirable, we must do it in such a way that we do not lose information with which the benchmarks can be related to classes of applications.

We also identify some steps that might be taken to improve the methodology. The development of more tools would be beneficial. A tool for evaluating the benchmark results would be useful. A program generator to generate the benchmark programs from the basic job models would also be helpful.

Another area of the methodology which requires further work is that of the multiple-user benchmarks. The model proposed in this thesis is based on the premise that it is reasonable to compare the multiple-user benchmark results to the single-user benchmark results. We have not validated this premise through actual experimentation. It will be interesting to discover whether the premise holds.

A final research direction is to use the knowledge gained from benchmarking in constructing analytical and simulation models. Combining the knowledge gained from benchmarks with analytic and numerical techniques may lead to the development of flexible, predictive performance evaluation tools for database machines and DBMSs. Such tools will be useful in the design of new systems as well as in the evaluation of existing systems. A performance evaluation model for comparative evaluation of database machine architectures is being developed at the University of Bremen [Schi83]. Integration of the knowledge gained from benchmarking with this type of model is what we propose.

APPENDIX A

THE RELATION GENERATOR

The Relation Generator is a tool for generating relations for a synthetic database. In this appendix, we present in pseudo-code design specifications for the relation generator. The pseudo-code style is PASCAL-like. The Relation Generator which we implemented for our experiments was written in PASCAL.

The relation generator is divided into two major sections. In the first section, the parameters which describe the relation and the attribute values in the relation are collected. It will be useful to implement this part of the relation generator as an interactive program, prompting the user for the required parameters. The second section of the generator actually generates the relation, using the parameters collected in the first section. All of the values for each attribute in turn are generated and stored in a temporary file. After all of the values for all of the attributes have been generated, the relation is generated by concatenating in proper sequence the attribute values from the temporary files.

All output will be character string output. Integer values will be converted to character strings of the appropriate length with a leading '+' or '-' in constructing records. We assume that these character strings can be formatted into the proper representation for storage in the database either by some database utility or some system utility. Values will be generated in the following modes:

1) Integers sequentially over a given range;

2) Integers pseudo-randomly over a given range;

3) Unique integers pseudo-randomly over a given range;

4) Characters in collating sequence for fixed and variable length strings;

5) Integers or characters selected pseudo-randomly from predefined sets of values;

6) Integers or characters selected according to a discrete distribution from predefined sets of values. The proportions will be limited to multiples of 5% of the number of tuples in a relation.

```
PROGRAM Generate_Relation

{Generate synthetic data for benchmarking
 purposes.  Data may be generated in the
 following modes:

 Generate
  Mode        Description

   1         Integers sequentially over a
             given range.
   2         Integers pseudo-randomly
             over a given range.
   3         Unique integers pseudo-randomly
             over a given range.
   4         Characters in collating sequence
             for fixed and variable length
             strings.
   5         Integers or characters selected
             pseudo-randomly from predefined
             sets of values.
   6         Integers or characters selected
             according to a discrete distribution
             from predefined sets of values.
             The proportions are limited to
             multiples of 5% of the number of
             tuples in the relation.
                                          }

CONST
   MaxNumberOfAttributes = ?; {Maximum number of
                               attributes in a
                               relation.}
   NameLength = ?;            {Length of attribute,
                               relation, file
                               names.}
```

```
TYPE
   NAME = Array [1 .. NameLength] of CHAR;

   {Information about attributes which will be
    required to generate data.  An array of
    these records will hold all required data
    for a relation. }
    ATTR_REC =
      RECORD
        Attr_Name        : NAME;
        Attr_Type        : CHAR;
        String_Length    : INTEGER;
        Lower_Bound      : INTEGER;
        Upper_Bound      : INTEGER;
        Generate_Mode    : CHAR;
        Seed             : INTEGER;
        Value_Set_Name   : FILENAME;
        Number_of_Values: INTEGER;
        Rel_Proportions : Array [1 .. 20] of
                                 INTEGER;
        Filename         : NAME
      END;

      ATTR_ARRAY =
         Array [1 .. MaxNumberOfAttributes]
                    of ATTR_REC;

VAR
   Relation_Name          : NAME;
   Buffer_Length          : INTEGER;
   Number_of_Tuples       : INTEGER;
   Number_of_Attributes : INTEGER;
   Attr_Info              : ATTR_ARRAY;

BEGIN

   {Get parameters required to define
    tuple template and generate attribute
    values.}
    Get_Parameters(Relation_Name,
                   Buffer_Length,
                   Number_of_Tuples,
                   Number_of_Attributes,
                   Attr_Info);
    {Generate tuples. }
     Generate_Data(Relation_Name,
                   Buffer_Length,
                   Number_of_Tuples,
                   Number_of_Attributes,
                   Attr_Info)

END Generate_Relation;
```

```
PROCEDURE Get_Parameters(Relation_Name : NAME,
                          Buffer_Length : INTEGER,
                       Number_of_Tuples : INTEGER,
                   Number_of_Attributes : INTEGER,
                              Attr_Info : ATTR_ARRAY);

VAR
   i: INTEGER;

BEGIN

   read(Relation_Name,
        Number_of_Tuples);

   Buffer_Length <- 0;
   i <- 0;

   WHILE more attributes DO;

      BEGIN

         i = i + 1;

         read(Attr_Info[i].Attr_Name);
         read(Attr_Info[i].Attr_Type);
         {C for character, I for integer.}

         IF Attr_Info[i].Attr_Type -> 'C' THEN
            read(Attr_Info[i].String_Length);

         ELSE {'I'}

            BEGIN

               read(Attr_Info[i].Lower_Bound);
               read(Attr_Info[i].Upper_Bound)

            END

            read(Attr_Info[i].Generate_Mode);
        verify that Generate_Mode and Attr_Type
           are compatible;

   {           Attr_  Generate_
     Type      Mode         1   2   3   4   5   6

     C                                  x   D   x

     I                      D   x   x       x   x

           'x' indicates compatible
           'D' indicates compatible, default }

        IF Attr_Info[i].Generate_Mode is '5' or '6' THEN

           BEGIN

           read(Attr_Info[i].Value_Set_Name,
                Number_of_values);

           IF Attr_Info[i].Generate_Mode is '6' THEN

              BEGIN

               FOR j <- 1 TO 20 DO;
                  Attr_Info[i].Rel_Proportions <- 0;

               read(Attr_Info[i].Rel_Proportions)

               END  {FOR}

           END;  {IF}

        Buffer_Length <- Buffer_Length +
                           length of character
                           representation of
                           current attribute

  END;  {WHILE}

  read(Number_of_Tuples;)
  Number_of_Attributes <- i;

END Get_Parameters
```

```
PROCEDURE Generate_Data(Relation_Name : NAME,
                        Buffer_Length : INTEGER,
                     Number_of_Tuples : INTEGER,
                 Number_of_Attributes : INTEGER,
                            Attr_Info : ATTR_ARRAY);

VAR
   i,j       : integer;
   Buffer    : Array [1 .. Buffer_Length] of CHAR;

BEGIN

   {Generate the values for each attribute
    in turn, storing the values in temporary
    files.}

   FOR i <- 1 TO Number_of_Attributes DO;

      BEGIN

      Attr_Info[i].Filename <- "temp" || char(i);

      CASE Attr_Info[i].Generate_Mode OF

        1:  Sequential_Int(Attr_Info[i],
                       Number_of_Tuples);

        2:  Random_Int(Attr_Info[i],
                       Number_of_Tuples);

        3:  Unique_Random_Int(Attr_Info[i],
                       Number_of_Tuples);

        4:  Char_String(Attr_Info[i],
                Attr_Info[i].String_Length,
                Number_of_Tuples);

        5:  Random_Values(Attr_Info[i],
                       Number_of_Tuples);

        6:  Discrete_Values(Attr_Info[i],
                       Number_of_Tuples)

     END   {CASE}
```

```
      END;    {FOR}

   {Build tuples by concatenating the values
    for each attributes from the temporary
    files.}

   open file Relation_Name;

   FOR i <- 1 TO Number_of_Attributes DO;

      open file Attr_Info[i].Filename;

   FOR j <- 1 TO Number_of_Tuples DO;

      BEGIN

      Buffer <- '';  {Buffer empty}

      FOR  i <- 1 TO Number_of_AttributesDO;

         BEGIN

         Buffer <- Buffer || next value from
                              Attr_Info[i].Filename;

         write(Relation_Name, Buffer)

         END  {FOR}

    END  {FOR}

END Generate_Data;
```

```
PROCEDURE Sequential_Int(Attr_Specs : ATTR_REC,
                         Number_of_Tuples : INTEGER);

VAR
   i,j,k,l : INTEGER;

BEGIN

   open file Attr_Specs.Filename;

   j <- 0;
   k <- Attr_Specs.Upper_Bound -
          Attr_Specs.Lower_Bound;

   FOR i <- 1 TO Number_of_Tuples DO;

      BEGIN

      l <- Attr_Specs.Lower_Bound + j;

      write(Attr_Specs.Filename, char(l));

      j <- (j + 1) MOD k

      END  {FOR}

   close file Attr_Specs.Filename

END Sequential_Int;
```

```
PROCEDURE  Random_Int(Attr_Specs : ATTR_REC,
                      Number_of_Tuples : INTEGER);

VAR
   i,j,k,l  : INTEGER;

BEGIN

  open file Attr_Specs.Filename;

  k <- Attr_Specs.Upper_Bound -
            Attr_Specs.Lower_Bound;

  FOR i <- 1 TO Number_of_Tuples DO;

     BEGIN

       {"random" is assume to be a random number
         generator, requiring a seed }

       j <- random(Attr_Specs.Seed) MOD k;

       l <- Attr_Specs.Lower_Bound + j;

       write(Attr_Specs.Filename, char(l))

     END  {FOR}

  close file Attr_Specs.Filename

END Random_Int;
```

```
PROCEDURE  Unique_Random_Int(Attr_Specs : ATTR_REC,
                             Number_of_Tuples : INTEGER);

VAR
   i,j,k,l : INTEGER;
   uniques :
       Array [Attr_Specs.Lower_Bound ..
                Attr_Specs.Upper_Bound] of INTEGER;

BEGIN

   FOR i <- Attr_Specs.Lower_Bound TO
            Attr_Specs.Upper_Bound DO;
      uniques[i] <- 0;

      open file Attr_Specs.Filename;
      k <- Attr_Specs.Upper_Bound -
            Attr_Specs.Lower_Bound;

      FOR i <- 1 TO Number_of_Tuples DO;
         BEGIN

           {"random" is assume to be a random number
             generator, requiring a seed }
           j <- random(Attr_Specs.Seed) MOD k;
           l <- Attr_Specs.Lower_Bound + j;

           WHILE uniques[l] = 1 DO;
              BEGIN

              l <- (l + 1) mod Attr_Specs.Upper_Bound;

              IF l -> 0 THEN
                 l <- Attr_Specs.Lower_Bound;

              END;  {WHILE}

           uniques[l] <- 1;
           write(Attr_Specs.Filename, char(l))

         END  {FOR}

      close file Attr_Specs.Filename

END Unique_Unique_Random_Int;
```

```
PROCEDURE Char_String(Attr_Specs : ATTR_REC,
                      String_Length : INTEGER,
                      Number_of_Tuples : INTEGER);

VAR
   i            : INTEGER;
   buildstring : array [1 .. String_Length]
                  of CHAR;

BEGIN

   buildstring[1] <= 'A';

   FOR i <- 2 TO String_Length - 2 DO;
       buildstring[i] <- ' ';

   open file Attr_Specs.Filename;

   FOR i <- 1 TO Number_of_Tuples DO;

      BEGIN

         write(Attr_Specs.Filename,
                        buildstring);

         {successor is a function which returns the
          next string in the collating sequence}
         buildstring <- successor(buildstring)

      END  {FOR}

   close file Attr_Specs.Filename

END Char_String;
```

```
PROCEDURE  Random_Values(Attr_Specs : ATTR_REC,
                    Number_of_Tuples : INTEGER);

VAR
   i,j : INTEGER;

BEGIN

   open file Attr_Specs.Filename;

   open file Attr_Specs.Value_Set_Name;

   FOR i <- 1 TO Number_of_Tuples DO;

      BEGIN

         j <- random(Attr_Specs.Seed) MOD
               Attr_Specs.Number_of_Values;

         write(Attr_Specs.Filename,
               jth value from file
               Attr_Specs.Value_Set_Name)

      END;  {FOR}

   close files Attr_Specs.Filename,
         Attr_Specs.Value_Set_Name;

END Random_Values;
```

```
PROCEDURE Discrete_Values(Attr_Specs : ATTR_REC,
                    Number_of_Tuples : INTEGER);

VAR
   i,j,sum : INTEGER;

BEGIN

   FOR j <- 1 TO 20 DO;
       Attr_Specs.Rel_Proportions[j] <-
          Attr_Specs.Rel_Proportions[j] *
          Number_of_Tuples / 100;

   sum <- 0;

   FOR j <- 1 TO 20 DO;
       sum <- sum +
             Attr_Specs.Rel_Proportions[j];

   IF sum < Number_of_Tuples THEN

      FOR j <- 1 TO (Number_of_Tuples - sum) DO;

      Attr_Specs.Rel_Proportions[1] <-
        Attr_Specs.Rel_Proportions[1] + 1;

   open file Attr_Specs.Filename;
   open file Attr_Specs.Value_Set_Name;

   FOR i <- 1 TO 20 DO;

      FOR j <- 1 TO
           Attr_Specs.Rel_Proportions[i] DO;

         write(Attr_Specs.Filename,
                  ith value from file
                  Attr_Specs.Value_Set_Name;

   close files Attr_Specs.Filename,
              Attr_Specs.Value_Set_Name

END Discrete_Values;
```

APPENDIX B

LIST OF BASIC BENCHMARKS

I. Unindexed Selection of 5% of Tuples

II. Selection of 5% of Tuples Qualified on Secondary-Key Attribute

III. Increasingly Qualified Selection with ANDed Predicates

IV. Increasingly Qualified Selection with ORed Predicates

V. Single-Tuple Selections Qualified on Integer, Primary-Key Attribute

VI. Single-Tuple Selections Qualified on Character, Primary-Key Attribute

VII. Unqualified Selection with Ordering on Non-Key Attribute

VIII. Unqualified Selection with Ordering on Primary-Key Attribute

IX. Single Aggregate with Grouping on Non-Key Attribute

X. Multiple Aggregates with Grouping on Non-Key Attribute

XI. Single Aggregate with Duplicates Eliminated and Grouping on Non-Key Attribute

XII. Projection of 25% of Attribute Values Qualified on Secondary-Key Attribute

XIII. Projection of 50% of Attribute Values Qualified on Secondary-Key Attribute

XIV. Projection of 75% of Attribute Values Qualified on Secondary-Key Attribute

XV. Equijoin on Non-Key Attribute

XVI. Equijoin on Primary-Key Attribute

XVII. Equijoin on Primary-Key Attribute with Alternative Placement Strategy

XVIII. Equijoin on Primary-Key Attribute with Join Predicate Reversed

XIX. Equijoin on Primary-Key Attribute with Selection on Secondary-Key Attribute

XX. Semijoin on Primary-Key Attribute with Selection on Secondary-Key Attribute

XXI. Semijoin on Primary-Key Attribute with Selection on Secondary-Key Attribute using ANY Set Operator

XXII. Inequality Semijoin on Non-Key Key Attribute with Selection on Secondary-Key Attribute

XXIII. 3-Way Equality Semijoin on Primary-Key Attribute with Selection on Secondary-Key Attribute

APPENDIX C: USING THE CENTRAL LIMIT THEOREM TO CHOOSE SAMPLE SIZE

The Central Limit Theorem (Lindeberg and Levy) for the Sample Mean. Let Φ denote the distribution function of the standard normal distribution.

If random variables $X_1, \ldots, X_n$ form a random sample of size n from a given distribution with a mean μ and a variance $\sigma^2 (0<\sigma^2<\infty)$, then for any fixed number x,

$$(1) \quad \lim_{n\to\infty} \left[\frac{n^{\frac{1}{2}}(\bar{X}-\mu)}{\sigma} \le x \right] = \Phi(x).$$

The interpretation of Eq. (1) is as follows: If a large random sample is taken from any distribution with mean μ and variance σ^2, regardless of whether this distribution is discrete or continuous, then the distribution of the random variable $n^{\frac{1}{2}}(\bar{X}_n-\mu)/\sigma$ will be approximately a standard normal distribution. Therefore, the distribution of $\bar{X}_n$ will be approximately a normal distribution with mean μ and variance σ^2/n or, equivalently, the distribution of the sum $\Sigma_{i-1}^{n} X_i$ will be approximately a normal distribution with a mean $n\mu$ and variance $n\sigma^2$ [DeGr75].

Example 1. What sample size is required to insure $Pr(|\bar{X}-\mu|\leq .5\sigma)=.95$?

$$Pr(|\bar{X}-\mu|\leq x\sigma) = Pr(n^{\frac{1}{2}}|\bar{X}-\mu|\leq xn^{\frac{1}{2}})$$

$$= 2\Phi(xn^{\frac{1}{2}})-1$$

Let $x = .5$ and $Pr(|\bar{X}-\mu|\leq .5\sigma) = .95$.

Then $Pr(|\bar{X}-\mu|\leq .5\sigma) = Pr(n^{\frac{1}{2}}|\bar{X}-\mu|\leq xn^{\frac{1}{2}})$

$$= 2\Phi(.5n^{\frac{1}{2}})-1,$$

and $2\Phi(.5n^{\frac{1}{2}})-1 = .95$

$$\Phi(.5n^{\frac{1}{2}}) = .975 = \Phi(1.96)$$

Therefore $.5n^{\frac{1}{2}} = 1.96$

$$n^{\frac{1}{2}} = 3.92$$

$$n \approx 16.$$

A sample size of 16 is required.

Example 2. What sample size is required to insure that $\Pr(|\bar{X}-\mu|\leq .25\sigma)=.95$?

Let $x = .25$ and $\Pr(|\bar{X}-\mu|\leq .25\sigma) = .95$.

$$\begin{aligned}
\text{Then } \Pr(|\bar{X}-\mu|\leq .25\sigma) &= \Pr(n^{\frac{1}{2}}|\bar{X}-\mu|\leq .25n^{\frac{1}{2}}) \\
&= 2\Phi(.25n^{\frac{1}{2}})-1
\end{aligned}$$

$$\begin{aligned}
\text{and} \qquad 2\Phi(.25n^{\frac{1}{2}}) &= .95 = \Phi(1.96) \\
\Phi(.25n^{\frac{1}{2}}) &= .975 = \Phi(1.96)
\end{aligned}$$

$$\begin{aligned}
\text{Therefore} \qquad .25n^{\frac{1}{2}} &= 1.96 \\
n^{\frac{1}{2}} &= 7.84 \\
n &\approx 62.
\end{aligned}$$

A sample size of 62 is required.

Example 3. Given a sample size of 30, what is the interval around the mean such that the sample mean is within the interval with probability .95.

$n = 30$, so

$$\Pr(|\bar{X}-\mu| \leq x\sigma) = \Pr(30^{\frac{1}{2}}|\bar{X}-\mu| \leq 30^{\frac{1}{2}}x)$$

$$= 2\Phi(30^{\frac{1}{2}}x)-1$$

Let $2\Phi(30^{\frac{1}{2}}x)-1 = .95$

Then $\Phi(30^{\frac{1}{2}}x) = .975 = \Phi(1.96)$

and $30^{\frac{1}{2}}x = 1.96 \quad 30^{\frac{1}{2}} \approx 5.4773$

$$x \approx .3578.$$

The interval is $\pm .3578\sigma$.

APPENDIX D

PERFORMING SELECTION BEFORE JOIN

When a query specifies a join, with selection predicates for one or both relations, is it better from a performance standpoint to perform the selections before performing the join? When a selection predicate on a primary- or secondary-key attribute is specified, it is always better to perform the selection first. This is easy to see. Both the volume of relevant data and the number of participating tuples are reduced by the selection. Therefore, both the time required to access the data and the number of operations required to perform the join are reduced.

However, let us consider the case where the selection predicate or predicates are on non-key attributes. We will show that it is always better to perform selection before join when the cardinality of the result relation, R, is greater than the sum of the cardinalities of the source relation, S, and the target relation, T. We will show this for three different join algorithms. In each case, the time required to do join before selection will be compared to the worst-case time required to do selection before join. The worst-case time for selection before join occurs when there are selection predicates for both relations.

First let us examine the case where a straightforward join algorithm is used. In the case where join is performed before selection, the time complexity of the join is O(C(S)*C(T)). The time complexity of the selection is O(C(R)), where R is the result relation from the join operation, and 0 <= R <= C(S)*C(T). If selection is performed before join, the time complexity of the selection is O(C(S)+C(T)). The time complexity of the join after selection is O(p1*C(S)*p2*C(T)), where p1 and p2 represent the fractions of the total number of tuples selected from the source and target relations respectively.

Now, let 'a' be the time required to access a tuple, 'j' be the time required to do one operation for join, and 's' the time required to apply the selection to one tuple. Let x=C(S) and y=C(T). Let JS be the time for join before selection. We can write

JS = a*(x+y) + j*x*y + s*C(R).

Let SJ be the time for selection before join. Let x'=p1*C(S), be the number of tuples selected from the source relation, and y'=p2*C(T) be the number of tuples selected from the target relation. We can write

SJ = (a+s)*(x+y) + j*x'*y'.

We want to determine when JS > SJ.

JS > SJ <=>

a*(x+y) + j*x*y + s*C(R) >
(a+s)*(x+y) + j*x'*y' if and only if

j*x*y + s*C(R) > s*(x+y)+ j*x'*y'

Another way of writing this is

j*x*y + s*C(R) > s*(x+y) + j*x'*y' <=>

JS > SJ

This is the <u>necessary</u> condition. Now, clearly x >= x' and y >= y', so that j*x*y >= j*x'*y'. Therefore

j*x*y + s*C(R) >= j*x'*y' + s*C(R),

and

j*x'*y' + s*C(R) > s*(x+y) + j*x'*y' =>

JS > SJ.

So that the sufficient condition for JS > SJ is C(R) > (x+y). That is, when the cardinality of the result relation is greater than the sum of the cardinalities of the source and target relations, it is always better to perform selection before join.

Next let us look at the case where the join algorithm is a sort-and-match algorithm. Let 'c' be the time required to do a comparison. The time required for the join before selection can be written as

$$JS = a(x+y) + c(x*\log x + y*\log y) + c(2(x+y)-1) + s*C(R)$$

$$SJ = (a+s)(x+y) + c(x'*\log x' + y'*\log y') + c(2(x'+y')-1).$$

Then the necessary condition for JS > SJ is

$$(x * \log x) + (y * \log y) + c*(2*(x+y) - 1) + s*C(R) >$$

$$c*((x' * \log x') + (y' * \log y)) + c*(2*(x'+y') -1) + s*(x+y).$$

And since x >= x' and y >= y', then the sufficient condition is

$$C(R) > (x+y).$$

Again, it is always better to perform selection before join when the cardinality of the result relation is greater than the sum of the cardinalities of the source and target relations.

Finally, let us examine the case where a hashing algorithm is used. Let 'h' be the time required to hash a tuple to a location in memory. Then the time required for join before selection can be written as

$$JS = (a+h)(x+y) + s*C(R).$$

The time for selection before join can be written as

$$SJ = (a+s)(x+y) + h(x'+y').$$

The proof is similar to those given above.

APPENDIX E

CARDINALITY OF THE RESULT RELATIONS FOR INEQUALITY JOINS IN BENCHMARK XXII

We will show that the cardinality of the result relation for an inequality join from Benchmark XXII is greater than the sum of the cardinalities of the source and target relations. The join is made over the primary key attribute. Let C(S) be the cardinality of the source relation and C(T) be the cardinality of the target relation. Because the values for the primary key attribute were generated sequentially, there are exactly min{ C(S),C(T) } matches when the primary key values of the tuple in the source relation, S, are compared to the primary key values of the tuples in the target relation, T. Then the cardinality of the result relation, R, for an inequality join on the primary key is

$$C(S)*C(T) - \min\{ C(S),C(T) \}.$$

The sum of the cardinalities of the source and target relations is

$$C(S) + C(T).$$

Let $C(S) > 2$ and $C(T) > C(S)$. Then

$$C(S)*C(T) - \min\{ C(S),C(T) \} =$$

$$C(S) * C(T) - C(S) =$$

$$(C(T) - 1) * C(S).$$

Then

```
     (C(T) - 1) * C(S) and C(S) > 2 =>
         (C(T) - 1) * C(S) > (C(T) - 1) * 2
```

and

```
     (C(T) - 1) * 2 and C(T) > C(S)    =>
         (C(T) - 1) * 2 >= C(S) * 2    <=>
                C(T) - 1  >= C(S) <=> TRUE.
```

So, we have shown that when the cardinality of the source relation is greater than 2, and the cardinality of the target relation is greater than the cardinality of the source relation, the cardinality of the result relation is greater than the sum of the cardinalities of the source and target relations.

APPENDIX F

CLUSTER ANALYSIS

The means for the access-intensive, processor-intensive, and overhead-intensive query pools were analyzed using the Johnson clustering method [John67]. The Johnson method is a hierarchical clustering method. Both single-link and complete-link analyses were run. The output of the analysis is voluminous. We have chosen to show here single-link and complete-link results for each query pool. The order of the mean in shown rather than the actual mean, i.e., 1 represents the smallest mean, 2 the next smallest, etc. The output shown is for the level of dissimilarity at which the number of clusters is the same as the number of clusters arrived at by inspection. The clusters developed by inspection are also shown for comparison.

The output indicates that a clustering analysis will give a more reliable clustering of the means, since the level of dissimilarity is the basis for merging clusters. The final choice of clusters should be based on the level of dissimilarity and the magnitude of the dissimilarity, as well as the choice of a reasonable number of clusters.

CLUSTERS FOR THE ACCESS-INTENSIVE QUERY POOL

```
Results of clustering by inspection :

cluster  1 is   1  2  3  4  5  6  7  8  9 10 11
cluster 12 is   12
cluster 13 is   13 14 15 16
cluster 17 is   17 18 19 20 21
cluster 22 is   22
cluster 23 is   23 24 25


Results of Johnson single-link clustering:

 at dissimilarity  80 the clustering is :
cluster  1 is   1  2  3  4  5  6  7  8  9 10 11
               12 13 14 15 16
cluster 17 is  17 18
cluster 19 is  19 20
cluster 21 is  21
cluster 22 is  22
cluster 23 is  23 24 25


Results of Johnson complete-link clustering:

 at dissimilarity 137 the clustering is :
cluster  1 is   1  2  3  4  5  6  7  8  9 10 11 12
cluster 13 is  13 14 15 16 17 18
cluster 19 is  19 20
cluster 21 is  21
cluster 22 is  22
cluster 23 is  23 24 25
```

CLUSTERS FOR THE PROCESSOR-INTENSIVE QUERY POOL

```
Results of clustering by inspection :

cluster  1 is   1  2  3  4  5
cluster  6 is   6  7  8  9
cluster 10 is  10 11 12 13 14 15 16
cluster 17 is  17 18 19 20 21 22
cluster 23 is  23 24 25 26
cluster 27 is  27 28 29
cluster 30 is  31 32 33
cluster 34 is  34 35


Results of Johnson single-link clustering:

 at dissimilarity  75 the clustering is :
cluster  1 is   1  2  3  4  5  6  7  8  9 10 11
               12 13 14 15 16
cluster 17 is  17 18 19 20 21 22
cluster 23 is  23 24 25 26
cluster 27 is  27 28 29
cluster 30 is  30 31
cluster 32 is  32 33
cluster 34 is  34
cluster 35 is  35


Results of Johnson complete-link clustering:

 at dissimilarity 190 the clustering is :
cluster  1 is   1  2  3  4  5  6  7  8  9
cluster 10 is  10 11 12 13 14
cluster 15 is  15 16 17 18 19 20 21 22
cluster 23 is  23 24 25 26
cluster 27 is  27 28 29
cluster 30 is  30 31 32 33
cluster 34 is  34
cluster 35 is  35
```

CLUSTERS FOR THE OVERHEAD-INTENSIVE QUERY POOL

```
Results of clustering by inspection :

cluster  1 is   1  2  3
cluster  4 is   4  5  6  7
cluster  8 is   8  9 10 11 12 13
cluster 14 is  14 15 16


Results of Johnson single-link clustering:

 at dissimilarity  26 the clustering is :
cluster  1 is   1  2  3  4  5  6  7
cluster  8 is   8  9 10 11 12 13
cluster 14 is  14
cluster 15 is  15 16


Results of Johnson complete-link clustering:

 at dissimilarity  63 the clustering is :
cluster  1 is   1  2  3  4  5  6  7
cluster  8 is   8  9 10 11 12 13
cluster 14 is  14
cluster 15 is  15 16
```

LIST OF REFERENCES

[Baer80] Baer, Jean-Loup, Computer Systems Architecture, Computer Science Press, 1980.

[Ball65] Ball, G. H, "Data Analysis in the Social Sciences: What About the Details", Proceedings of the AFIPS Joint Computer Conference, 1965.

[Banc83] Bancilhon, F., D. Fortin, S. Gamerman, J. M. Laubin, P. Richard, M. Scholl, D. Tusera, A. Verroust, "VERSO: A Relational Back-End Database Machine", in Advanced Database Machine Architectures, David K. Hsiao, ed., Prentice-Hall, 1983.

[Bann78a] Bannerjee, Jayanta, and David K. Hsiao, "The Use of a Database Machine for Supporting Relational Databases", Proceedings of the 5th Annual Workshop on Computer Architecture for Non-Numeric Processing, August 1978.

[Bann78b] Bannerjee, Jayanta, and David K. Hsiao, "A Methodology for Supporting Existing Codasyl Databases with New Database Machines", Proceedings of the ACM '78 Conference, December, 1978.

[Bann78c] Bannerjee, Jayanta, Richard I. Baum, and David K. Hsiao, "Concepts and Capabilities of a Database Computer", ACM Transactions on Database Systems, Vol. 3, No. 4, December 1978.

[Bann79] Bannerjee, Jayanta, David k. Hsiao, and Krisnamurthi Kannan, "DBC - A Database Computer for Very Large Databases", IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979.

[Bann80] Bannerjee, Jayanta, David K. Hsiao, and Fred K. Ng, "Database Transformation, Query Translation, and Performance Analysis of a New Database Computer in Supporting Hierarchical Database Management", IEEE Transactions on Software Engineering, SE-6, No. 1, January 1980.

[Blas75] Blashfield, R. K., "A Consumer Report on Cluster Analysis Software: (4) USEABILITY", Technical Report, Department of Psychology, The Pennsylvania State University.

[Blas77] Blasgen, M. W., and K. P. Eswaran, "Storage and Access in Relational Data Bases", IBM Systems Journal, Vol. 4, 1977.

[Bodg83a] Bogdanowicz, Robert A., "Benchmarking the Selection and Projection Operations, and Ordering Capabilities of Relational Database Machines", Masters' Thesis, Naval Postgraduate School, Monterey, California, June 1983.

[Bogd83b] Bogdanowicz, R., M. Crocker, D. K. Hsiao, C. Ryder, V. Stone, P. Strawser, "Experiments in Benchmarking Database Machines", in Database Machines, H.-O. Leilich and M. Missikoff, ed., Springer-Verlag, 1983.

[Brit83a] Britton Lee, Inc., Intelligent Databae Machine Product Description, 1983.

[Brit83b] Britton Lee, Inc., IDM Software Reference Manual, Version 1.4, January 1983.

[Cham74] Chamberlin, D. D., and R. F. Boyce, "SEQUEL: A Structured English Query Language", Proceedings of the ACM SIGMOD Workshop on Data Description, Access, and Control, May 1974.

[Chri81] Christodoulakis, Stavros, "Estimating Selectivities in Data Bases", Technical Report CSRG-136, Compute Research Group, University of Toronto, 1981.

[Croc83] Crocker, Michael D., "Benchmarking the Join Operations of Relational Database Machines", Masters' Thesis, Naval Postgraduate School, Monterey, California, June 1983. [DeGr75] DeGroot, Morris H., Probability and Statistics, Addison-Wesley, 1975.

[DeWi79] Dewitt, David J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database management Systems", IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979.

[Dube80] Dubes, Richard and A. K. Jain, "Clustering Methodologies in Exploratory Data Analysis", Advances in Computers, Vol. 19, Academic Press, 1980.

[Epst80] Epstein, Robert and Paula Hawthorn, "Design Decisions for the Intelligent Database Machine", Proceedings of the National Computer Conference, 1980.

[Ferr78] Ferrari, Domenico, Computer Systems Performance Evaluation, Prentice-Hall, 1978.

[Ferr83] Ferrari, Domenico, Giuseppe Serazzi, and Alessandro Zeigner, Measurement and Tuning of Computer Systems, Prentice-Hall, 1983.

[Flyn66] Flynn, Michael J., "Very High-Speed Computing Systems", Proceedings of the IEEE, December 1966.

[Forg65] Forgy, E., "Cluster Analysis of Multivariate Data: Efficiency Versus Interpretability of Classifications", abstract, Biometrics, Vol 21.

[Free79] Freeman, Peter, "A Perspective on Requirements Analysis and Specification", Proceedings, IBM Design '79 Symposium.

[Gard83] Gardarin, Georges, Philippe Bernadat, Nicole Temmerman, Patrick Valduriez, Yann Viemont, "SABRE: A Relational Database System for a Multi-Microprocessor Machine", in Advanced Database Machine Architectures, David K. Hsiao, ed., Prentice-Hall, 1983.

[Hawt79] Hawthorn, P. B. and M. Stonebraker, "Performance Analysis of a Relational Data Base System", Proceedings of the ACM SIGMOD Conference, 1979.

[Hawt81] Hawthorn, P., "The Effect of Target Applications on the Design of Data Base Machines", Proceedings of the ACM SIGMOD Conference, 1981.

[Hawt82] Hawthorn, P. B., and D. J. DeWitt, "Performance Analysis of Alternative Database machine Architectures", IEEE Transactions on Software Engineering, Vol. SE-8, No. 1, January 1982.

[Hayn82] Haynes, Leonard S., "Database Machines Viewed as High Level Language Computers", Proceedings of the International Workshop on High-Level Language Computer Architecture, 1982.

[He83] He, Xin-Gui, Masanobu Higashida, David K. Hsiao, Douglas S. Kerr, Ali Orooji, Zhong-Zhi Shi, and Paula R. Strawser, "The Implementation of a Multi-Backend Database System (MDBS): Part II - The Design of a Prototype MDBS", in Advanced Database Machine Architectures, David K. Hsiao, ed., Prentice-Hall, 1983.

[Hsai82] Hsiao, David K., "Database Computers - A Tutorial and Review", unpublished, April 1982.

[Hsai83] Hsiao, David K., "Cost-Effective Ways of Improving Database Computer Performance", Proceedings of the National Computer Conference, 1983. [John67] Johnson, S. C., "Hierarchical Clustering Schemes", Psychometrika, Vol. 32, 1967.

[Mala82] Malabarba, Frank J., "Data Base Machines - It's About Time!", Department of the Navy.

[Meno81] Menon, M. Jaishankar, "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth", Ph.D. Dissertation, The Ohio State University, 1981.

[Miss83] Missikoff, M., M. Terranova, "The Architecture of DBMAC, A Relational Database Computer", in Advanced Database Machine Architectures, David K. Hsiao, ed., Prentice-Hall, 1983.

[Morr82] Morris, Michael F. and Paul R. Roth, Computer Performance Evaluation Tools and Techniques for Effective Analysis, Van Nostrand Reinhold Company, 1982.

[Ozka75] Ozkaharahan, E. A., S. A. Schuster, and K. C. Smith, "RAP — An Associative Processor for Data Base Management", Proceedings of the National Computer Conference, 1975.

[Rose76] Rosen, Saul, Lectures on the Measurement and Evaluation of the Performance of Computing Systems, Society for Industrial and Applied Mathematics, 1976.

[Ryde83] Ryder, Curtis M., "Benchmarking Relational Databae Machines' Capabilities in Supporting the Database Administrators' Functions and Responsibilities", Masters' Thesis, Naval Postgraduate School, Monterey, California, June 1983.

[Saue81] Sauer, Charles H. and K. Mani Chandy, Computer Systems Performance Modeling, Prentice-Hall, 1981.

[Schi82] Schill, John, "Relational DBMS Evaluation Report", reported at the ninth meeting of the Navy Data Base Machine Working Group, Monterey, California, October 1982.

[Schi83] Schiffner, G., P. Scheuermann, S. Seehusen, H. Weber, "On a Specification and Performance Evaluation Model for Multicomputer Database Machines", Database Machines, Springer-Verlag, 1983.

[Schu79] Schuster, Stewart A., H. B. Nguyen, Esen A. Ozkarahan, and Kenneth C. Smith, "Rap.2 — An Associative Processor for Databases and Its Applications", IEEE Transactions on Computers, Vol., C-28, No. 6, June 1979.

[Schw83] Schweppe, H., H. Ch. Zeidler, W. Hell, H.-O. Leilich, G. Stiege, W. Teich, "RDBM - A dedicated Multiprocessor System for Data Base Management", in Advanced Database Machine Architectures, David K. Hsiao, ed., Prentice-Hall, 1983.

[Seli79] Selinger, P. G., M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System", IBM Research Report RJ2429, 1979.

[Smit79] Smith, Diane C. P. and John Miles Smith, "Relational Data Base Machines, Computer, March 1979.

[Ston83] Stone, Vincent C., "Design of Relational Database Benchmarks", Masters' Thesis, Naval Postgraduate School, Monterey, California, June 1983.

[Su79a] Su, Stanley Y. W., "Cellular-Logic Devices: Concepts and Applications", Computer, March 1979.

[Su79b] Su, Stanley Y. W., Le Huu Nguyen, Ahmet Emam, and G. J. Lipovski, "The Architectural Features and Implementation Techniques of the Multicell CASSM", IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979.

[Tana83] Tanaka, Yuzuru, "A Data Stream Database Machine with Large Capacity", in Advanced Database Machine Architectures, David K. Hsiao, ed., Prentice-Hall, 1983.

[Wong76] Wong, E. and K. Youssefi, "Decomposition - A Strategy for Query Processing", ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976.

[Yao78] Yao, S. B. and D. DeJong, "Evaluation of Database Access Paths", Proceedings of the ACM SIGMOD Conference, 1978.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93943	40
Dr. Paula R. Strawser Code 52Xw Department of Computer Science Naval Postgraduate School Monterey, CA 93943	15
Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217	1
Dr. David W. Mizell Office of Naval Research 1030 East Green Street Pasadena, CA 91106	1
Prof. Karsten Schwan The Ohio State University Department of Computer and Information Science 2036 Neil Avenue Columbus, OH 43210	1
Ms. Verlynda Dobbs 2545 Kenmcunt Court Beaver Creek, OH 45385	1
LT Robert Bogdanowicz 212 Worden Street Newport, RI 02840	1

LT Michael Crocker 428 Warley Street Newport, RI 02840	1
LCDR Vincent C. Stone 114 Mallard Drive Groton, CT 06340	1
LCDR Curtis Ryder 9905 Lamott Court Ellicott City, MD 21043	1
CDR T. M. Pigoski HQ COMNAVSECGRU G30 D 3801 Nebraska Ave NW Washington, D.C. 20390	1
Ms. Doris Mleczko DPSCWEST Code 0340 Pacific Missile Test Center Pt. Mugu, CA 93042	3
Ms. Gwen Hunt Director, Plans and Management Code 0101 Pacific Missile Test Center Pt. Mugu, CA 93402	1
Dr. K. I. Lichti Technical Director Code 02 Pacific Missile Test Center Pt. Mugu, CA 93042	1
Mr. Stephen Fuld Amperif Corporation 21345 Lassen Street Chatsworth, CA 91311	1
Mr. Mike Lombard Amperif Corporation 21345 Lassen Street Chatsworth, CA 91311	1
Mr. Dave Clements Manager, Product Marketing Teradata Corporation 6383 Arizona Circle Los Angeles, CA 90045	1
Mr. Mike Ubell Manager, Software Development Britton Lee, Inc. 1919 Addison Street #304 Berkeley, CA 94704	1

Dr. John Schill Naval Ocean Systems Center Code 832 San Diego, CA 92152	1
Dr. Ali Orooji Department of Computer and Information Science The Ohio State University 2036 Neil Avenue Columbus, OH 43210	1
Dr. Walter Giffin Department of Industrial and Systems Engineering 1971 Neil Avenue Columbus, OH 43210	1
Mr. Glenn Meyer 982 Henderson Avenue, Apt. 4 Sunnyvale, CA 94086	1
Dr. Lee J. White Department of Computing Science University of Alberta Edmonton, Alberta T6G 2H1 CANADA	1
Mr. Dave Herman Accuray Corporation 650 Ackerman Road Columbus, OH 43202	1
Mr. Carl Hudgins DLA Training Center Data Systems Automation Center c/o Defense Construction Supply Center 3900 E. Broad Street Columbus, OH 43215	2
Mr. Jim Starkey DEC Database Group MK1-2/J12 Digital Equipment Corporation Continental Boulevard Merrimack, NH 03054	1
Dr. Dina Bitton Computer Science Department Cornell University 405 Upson Hall Ithaca, NY 14853	1
Herr Gerhard Schiffner Einsteinufer 61 D-1000 Berlin 10 WEST GERMANY	1

Herr Norbert Vorstadt 1
GEI-Gesellschaft
Albert-Einstein-Str. 61
5100 Aachen-Walheim
WEST GERMANY

Fr. Charlotte Dieter 1
Institut fur Theoretische und
Praktische Informatik
Gaußstrasse 11
D-33 Braunschweig
WEST GERMANY

Dr. Ellen Oliver 1
Digital Equipment Corporation
301 Rockrimmon Blvd. S.
Colorado Springs, CO 80919

NUWES 1
Code 03A Bldg. 94
ATTN: Mike DeVaney
Keyport, WA 98345

Dr. M. Jaishankar Menon 1
IBM Corporation
K61/282
5600 Cottle Road
San Jose, CA 95193

Prof. Fred Maryanski 1
University of Connecticut
Dept. of Electrical Engineering
& Computer Sciences, U-157
Storrs, CT 06268

Prof. David DeWitt 1
Dept. of Computer Sciences
University of Wisconsin
Madison, WI 53706

DAT
FILM

Los Angeles, CA 90045

Mr. Mike Ubell
Manager, Software Development
Britton Lee, Inc.
1919 Addison Street #304
Berkeley, CA 94704

1

Dr. Dina Bitton Computer Science Department Cornell University 405 Upson Hall Ithaca, NY 14853	1
Herr Gerhard Schiffner Einsteinufer 61 D-1000 Berlin 10 WEST GERMANY	1